

University of Alberta

A Two Tiered Approach to Methodical GUI Testing

By

Ping Li



A thesis submitted to the Faculty of Graduate Studies and Research in
partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta

Fall 2004



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

ISBN: 0-612-95798-5

Our file *Notre référence*

ISBN: 0-612-95798-5

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

CONTENT

1. SOFTWARE TESTING AND GUI TESTING.....	1
1.1. SOFTWARE TESTING	1
1.1.1. <i>Introduction.....</i>	<i>1</i>
1.1.2. <i>Software Testing Process.....</i>	<i>2</i>
1.2. GUI TESTING.....	4
1.2.1. <i>Evolution of User Interface.....</i>	<i>4</i>
1.2.2. <i>GUI Testing Challenges.....</i>	<i>6</i>
1.2.3. <i>Related Research on GUI Testing.....</i>	<i>9</i>
2. A TWO TIERED GUI TESTING STRATEGY	13
2.1. OVERVIEW	13
2.2. WHY WE NEED A TWO-TIERED APPROACH	16
2.3. INTRODUCTION TO THE EXAMPLE GUI SYSTEM USED IN THE RESEARCH.....	21
2.4. RESPONSIBILITY IDENTIFICATION AND EXAMPLE	22
2.4.1 <i>The Responsibility Definition and the Identification Criteria</i>	<i>23</i>
2.4.1.1 Criterion 1	23
2.4.1.2 Criterion 2	24
2.4.1.3 More Discussion on Responsibility	25
2.4.2 <i>Identify Responsibilities.....</i>	<i>27</i>
2.4.2.1 Identify Responsibilities with Software Requirement Specification ...	29
2.4.2.2 Identifying Responsibilities within the Design Stage	30
2.4.2.3 Identify Responsibilities within Implementation Stage	35
2.5. EFFORTS REQUIRED FOR THE PROPOSED APPROACH	37
3. COMPONENT TIER TESTING.....	43

3.1. MODELS OF GUI OBJECTS	43
3.1.1. <i>Examples of GUI Object Model</i>	43
3.1.2. <i>Input Analysis and the GUI Object Model</i>	47
3.2 FLOW GRAPH ON THE COMPONENT TIER	49
3.3 AN EXAMPLE OF A FLOW GRAPH.....	52
4. SYSTEM TIER TESTING.....	59
4.1. INTRODUCTION	59
4.2. FINITE STATE MACHINE.....	60
4.3 THE MODEL FOR THE SYSTEM TIER TESTING.....	65
5. COVERAGE CRITERIA	73
5.1. INTRODUCTION	73
5.2. COVERAGE CRITERIA OF GUI TESTING	73
6. EMPIRICAL INVESTIGATIONS AND RESULTS	78
6.1 OVERVIEW	78
6.2 EMPIRICAL INVESTIGATION 1	78
6.2.1 <i>Overview</i>	78
6.2.2 <i>Mutant Operators</i>	81
6.2.3 TEST EXECUTION RESULTS AND ANALYSIS	83
6.3 EMPIRICAL INVESTIGATION 2.....	89
6.3.1 <i>Introduction</i>	89
6.3.2 <i>Results and Discussion</i>	91
7. CONCLUSION AND FUTURE WORKS.....	94
BIBLIOGRAPHY	96
APPENDIX A	101
APPENDIX B	106

List of Tables

Table 2.1 A event and response table example	33
Table 2.2 Part of the Notepad Function List	34
Table 2.3 Major Testing Activities	40
Table 3.1 Test Cases for Print Range	49
Table 3.2 Summary of the Attributes for "Create a new file"	55
Table 6.1 The Mutant Operators Used in the Investigation	82
Table 6.2 The Result of The Experiment	84
Table 6.3 The Results of the Experiment	86
Table 6.4. The Total Number of the Event-sequences in Notepad	88
Table 6.5. The Percentage of the Event-sequences Covered	89

List of Figures

Figure 1.1 A standard commercial capture/reply tool	10
Figure 2.1 A Bug in Microsoft Word 2000	19
Figure 2.2 The GUI of Notepad	22
Figure 2.2 A Typical HELP Menu	26
Figure 2.3 A Sub-window of WinCvs	27
Figure 2.5 Developing Process	28
Figure 2.6 A Fragment of the SRS	30
Figure 2.7 A Fragment of a User Interface Specification	32
Figure 2.8 Major Phases of Software Testing	39
Figure 3.1 The Model of a Push Button	44
Figure 3.2 The Model of a Button with Enabled/Disabled State	44
Figure 3.3 The Model of a Radio Button	45
Figure 3.4 The Model of a Check Box	45
Figure 3.5 The Model of a List	46
Figure 3.6 The Model of a Editable Combobox	46
Figure 3.7 The Model of a Scroll Bar	47
Figure 3.8 The Print Dialog of Notepad Application	47
Figure 3.9 The Model of Print Range Radio Buttons	49
Figure 3.10 The Confirm Save Message Box	53
Figure 3.11. The Save a File Dialog	54
Figure 3.12 New Component Flow Graph	56
Figure 3.13 The Sub-graph of Save a File Dialog	57
Figure 4.1 Interactions among Components	59

Figure 4.2 An Example of a State Diagram	63
Figure 4.3 A Mealy Model Example	64
Figure 4.4 A Moore Model Example	64
Figure 4.5 A Typical State Transition Diagram	66
Figure 4.6 The Event Sequence Diagram	67
Figure 4.7 Event Sequence Diagram of "Open File"	68
Figure 4.8 Fault Event Sequence Diagram of "Open File"	69
Figure 4.9 The Event Sequence diagram of Notepad	71

1. Software Testing and GUI Testing

1.1. Software testing

1.1.1. Introduction

Software testing has never had as much appeal as other areas of computer science. It has not been treated as a serious topic until late 60's when the "software crisis" spawned the concept of software engineering (Roper[1994]). There is no way to guarantee that a software system is implemented perfectly. Software failures may come from many unexpected directions. In safety critical systems, certain defects can cause very serious results. In Leveson & Turner [1993], it was reported that a computerized radiation therapy machine called Therac-25 caused six known incidents of accidental overdoses between June 1985 and January 1987, which resulted in deaths and serious injuries. Today, more and more people have been aware of the significance of software testing and have contributed to its development. Varied approaches for testing have become matured with formal testing methods and techniques being adopted by testing professionals. In the preface of Myer [1979], it was indicated that approximately 50% or more of the software production development cost was spent on testing.

There are many published definitions of software testing and all of them boil down to the same thing: testing is a process of designing, preparing and executing tests in order to measure and improve the quality of the software being tested. However, it is challenging to thoroughly test even the simplest software system. Hence, varied techniques are used to design effective and efficient test cases to reveal faults in software systems. Here, a **test case** specifies the pre-test state of the software and its environment, the test inputs or conditions, and the expected result produced by the software application. A **test strategy** is an algorithm or heuristic to create test cases from a representation, an implementation, or a test model. A **test model** represents relationships among elements within a representation or implementation of a software system (Binder [2000]).

According to Fewster & Graham [1999], the quality of a test case can be described by the following attributes:

- **Effectiveness:** shows the detection ability of a test case to find defects.
- **Exemplariness:** an exemplary test case should test more than one thing so that the total number of test cases required should be reduced.
- **Cost considerations:** how economical a test case is to perform, analyze and debug; and how evolvable it is, for example, how much maintenance effort is required on the test case each time the software changes.

A well-designed test case requires that these attributes must be balanced one against another. For example, a high measure on the exemplariness which cost a lot to perform, analyze, debug and may require a lot of maintenance, and can result in low measure on the economic and evolvable scales (Fewster & Graham, [1999]).

Software testing is often used in association with the other two terms: verification and validation. Verification is the checking or testing of items, including software, for conformance and consistency with an associated specification. Software testing is just one kind of verification, which also uses techniques such as reviews, analysis, inspections and walkthroughs. Validation is the process to make sure that what has been specified is what the user actually wants (IPL [1997]).

1.1.2. Software Testing Process

In developing a large system, the testing process is applied to software applications at various stages throughout their life cycle. These stages, according to Watkins [2001], Kaner, Falk & Hung [1999] and Binder [2000], are:

- **Unit testing** is the lowest level of testing. Individual components are tested to ensure that they meet the requirements. Unit testing is done in a controlled environment whenever possible, so that the test team can feed a predetermined set of data to the component being tested and observe what output actions and data are produced. In practice, unit testing is conducted by the development team.
- **Integration testing** is the next step to ensure that the modules, which comprise the application under testing, interact together in a correct, stable, and coherent manner. Integration testing is the process of verifying that the system components work together as described in the system and program design specifications. The

development team typically conducts Integration Testing. Occasionally in large organizations with well-established quality and/or mature software development and testing processes, Integration Testing will be conducted by a dedicated *Testing Team*, which will be responsible for all aspects of the testing.

- **System testing**, in which the software is integrated into a complete product and tested, shows that all requirements are met and establish confidence that the software will be accepted by the users. System Testing is typically conducted as an independent testing project by a dedicated testing team.

- **Acceptance Testing** is to confirm that the system meets its business requirements and to provide confidence that the system works correctly and is usable before it is formally "handed over" to the users. Usually, it is performed by nominated user representatives under the guidance and supervision of the testing team.

- **Regression Testing** is to provide confidence that the software still functions correctly following modification or extension of the system (such as user enhancements or upgrades or following new builds or releases of the software). It is not strictly speaking a testing phase, but is a testing technique that can be applied to any of the other testing phases.

Within each level of testing, once the tests have been applied, test results are evaluated. If a problem is encountered, then either the tests are revised and applied again, or software is fixed and the tests are applied again. This is repeated until there is no problem. Then the development can proceed to the next level of testing. In today's view, testing is not a phase of development after coding any more. It is performed in parallel with the software effort and consists of its own phases of analysis (planning and setting test objectives and test requirements); design (specifying the tests to be developed); implementation (constructing or acquiring the test procedures and cases); execution (running and rerunning the tests); and maintenance (saving and updating the tests as the software changes). This life cycle perspective of testing represents a dramatic change from just a few years ago, when almost everyone equated testing with execution. The activities of planning, designing, and constructing tests were not recognized, and testing wasn't seen as *starting* until you started running tests (Hetzel [1998]).

1.2. GUI Testing

Graphical user interfaces (GUIs) are more common in software applications than ever. Their function is to guide users through their interactions with the system, presenting appropriate responses from the underlying application according to the users' actions. In the marketplace, many special software systems and tools are devoted to helping with the design and implementation of GUIs. These, in turn, have helped to produce a new generation of GUIs that are easier to use than ever before. At the same time, however, GUIs are becoming larger and more complex. They can account for over half of the code and development time for many software systems (Myers [1993]). Testing GUIs for correctness can enhance an entire system's usability, security, and robustness. But, unfortunately, testing a GUI may be very difficult and challenging, because it has many characteristics that differ from traditional software, and, hence, conventional code-based testing techniques cannot always be directly applied to GUIs testing.

1.2.1. Evolution of User Interface

User interfaces refer to the methods and devices that are used to accommodate interaction between machines and the human beings who use them. The rest part of a software program is called the application, which implements the functions defined in the specification. User interfaces can take on many forms, but they always accomplish two fundamental tasks: communicating information from the machine to the user and communicating information from the user to the machine. The following briefly describes the history of user interfaces (Galitz [2002]).

The very first computers had user interfaces that were as rudimentary as the computers themselves. The computer communicated information to the user through flashing lights, and the user communicated information to the computer by setting mechanical switches. Only highly trained specialists were able to communicate directly with a computer.

In the next stage of evolution, computers communicated to users through printing devices, and the users communicated to the computer through punch cards. This was

an improvement, but was still cumbersome and inefficient. It was still rare for anyone but a computer specialist to actually communicate directly with a computer.

User interfaces left the dark ages when video screens were used to communicate information from the computer to the user, and typewriter-style keyboards were used to communicate information directly to computers. This major innovation helped to allow ordinary users to communicate directly with computers. But since video screens were limited to displaying only the characters that were found on the keyboard, the usefulness of the user interface was constrained by the same limitation. Users were required to memorize commands that were generally tailored more to the computers than to the users. A great deal of training was required before anyone could use a computer.

The first GUI ever developed was the work of Dr. Douglas Englebart, a researcher at Stanford Research Institute in 1960s. His visionary and pioneering design and prototypes succeeded in producing the world's first screen-based windows, cursor-selectable pop-up menus, as well as the mouse with which to interact with them. However, it was not until a decade later that researchers at the Xerox Palo Alto Research Center began to break away from the character-based interface paradigm and invented the first Graphic User Interface on the Alto system. Most elements of the GUI that are very familiar today: windows, icons, menus, etc. existed in this first prototype. The use of graphics to communicate information to users visually in addition to textually makes the computer more simple and natural to use. And GUIs present a finite number of options to users rather than requiring users to memorize and manually enter commands from a virtually unlimited set of options. In this way, the interface was focused on the needs of human beings rather than the other way around. This significantly reduced the training that was necessary to use a computer, and for the first time uninitiated users were able to become productive almost immediately.

The expansion of the World Wide Web since the early 1990s has been truly amazing. Once simply a communication medium for scientists and researchers, its many and pervasive tentacles have spread deeply into businesses, organizations, and homes around the world. Unlike earlier text-based and GUI systems that were developed and nurtured in an organization's Data Processing and Information Systems groups, the

Web's roots were sown in a market-driven society thirsting for convenience and information. While the introduction of the graphical user interface revolutionized the user interface, the Web has revolutionized computing. It allows millions of people scattered across the globe to communicate, access information, publish, and be heard. It allows people to control much of the display and the rendering of Web pages. Aspects such as typography and colors can be changed, graphics turned off, and decisions made whether or not to transmit certain data over non-secure channels or whether to accept or refuse cookies. Nowhere in the history of computing has the user been given so much control.

A Web interface has many similarities with GUIs. They are both used by people, are interactive, are heavily visual experiences presented through screens, and are composed of many similar components. However, significant differences between them do exist. The characteristics of web interfaces cause that its design and testing are different from GUI testing in many aspects. This thesis is only focused on GUI testing.

1.2.2. GUI Testing Challenges

Today, many user interface development tools are devoted to helping with the design and implementation of GUIs. They are of considerable benefit to developers. Because they release the developer from the concerns of interface design in most environments, GUI design standards impose conventions which make one application look very much like another on the same platform.

However, for testers, testing GUIs can be a nightmare because a number of their special characteristics cause most traditional testing strategies, testing evaluation standards and practical testing application techniques cannot be applied to GUI testing directly. Below we summarize some major reasons for this problem (Gerrard [1997], Memon [2001], and White [2002]):

Event-driven software and extremely large input space

The event-driven nature of GUIs presents the first serious testing difficulty. GUI software enables users to click on any pixel on the screen at any time. The input to the system is actually a set of event sequences. Users have an extremely wide choice of

actions. At any point in an application, users may click on any field or object within a window. The permutations of these events can be huge. Furthermore, today, GUIs are typically developed by using GUI objects provided in advanced toolkits. Many GUI objects require inputs from users. Although these objects usually have default values, users may choose to input values by clicking on a selection or entering a data item directly. The permutations of all these input values grow dramatically when the GUI system is complex. Consequently, the extremely large size of different inputs and events leads to a large number of GUI states that need to be tested.

Another issue with event-driven nature is that in more advanced development environments, where sophisticated frameworks are being used, many of these events are handled "behind the scenes". With less advanced toolkits, the programmer must write code to handle these events explicitly. Many errors occur because the programmer cannot anticipate all of the contexts in which their event handlers are invoked. On the other hand, when testers are presented with a GUI application to test, it is very hard for them to dig out all the hidden complexities in the simplicity.

Hidden synchronization and dependencies

It is very common for GUI objects to have some forms of synchronization and dependency implemented. For example, if a check box is set to true, a text box intended to accept a numeric value elsewhere in the window may be made inactive or invisible.

In many situations, the synchronizations and dependencies between objects cannot be restricted to objects in the same window. For example, a visible window may present customer details including, say, "date of last order". Another open window may be used to log customer orders, so if the user creates and confirms an order for the same customer, should the "date of last order" field on the first window be updated? Most users would suggest that it should be, so the programmer must use the event handling mechanisms to implement the synchronization functionality. The concern for the tester is discovering these hidden relationships. However, finding all these relationships is not easy.

Object oriented software programming

Today's GUIs map very well to the object-oriented paradigm. The significant challenges for object-oriented software testing can be met in GUI testing. And during development, a developer typically uses instances of precompiled objects stored in a library to build the GUIs hierarchically. The source code for these elements may not always be available for code-based coverage evaluation. So traditional coverage criteria cannot be applied to GUI testing directly.

Many ways in, many ways out

One of the advantages of GUIs is that they can provide multiple ways for users to interface with an application. In the majority of situations in GUI applications, the user may use a keyboard shortcut, a button click, a menu option, a click on another window, etc. How many of these should be tested?

For most options within GUI applications, there are three ways of implementing functionality: keyboard shortcuts, function keys, and mouse movements (such as click on buttons or menus). Given that these three mechanisms are available for many options, does this mean that we must test these features three times?

Window management

In a GUI environment, users employ the standard features of the window management system, which is a software package to help users monitor and control different contexts by separating them physically: onto different parts on one or more display screens (Myers [1995]). These features include window movement, resizing, maximization, minimization, and closure. The programmer can determine which standard window controls are available. Although the operating system handles the windows' behaviors, the programmer must anticipate the impact of these behaviors on the application.

For example, closing a window before completing a transaction may leave the application or the database in an inconsistent state. Programmers might avoid such complications by disabling all of the standard window buttons and commands. However, they might also have made it impossible for users to reverse or undo certain actions. From the tester's point of view, which standard window controls need to be

tested? Where is the dividing line between testing the application and testing the operating system? Do we need to test navigation paths both forwards and backwards?

When analyzing the various challenges within GUI testing, we choose a heuristic to organize the GUI testing in a tiered fashion. On different tiers, we target different testing concerns of the GUI. The principle of our approach in this thesis is that we test the lowest level of detail in the components first. We then implement the tests for the entire application. In this way, we can build the testing up in trusted layers.

1.2.3. Related Research on GUI Testing

In practice, GUI testing currently relies on record - playback techniques to create and maintain test cases. With this method, one performs sample interactions with the GUI, giving it several different types of mouse and keyboard events. The tool records these events, usually as a script, and the tester later plays back the recorded sessions to recreate the events with different inputs (The. [1992]). Figure 1.1 shows the structure of a typical commercial capture/replay tool.

This approach has several major deficiencies, for example, the context sensitivity and synchronization problem (Kepple [1994]). Hence, Kepple proposed that in the GUI paradigm, testing should be event driven and focused at the level of logical object functionality. Furthermore, even with this computer-aided tool, GUI testers often find it impractical to generate all possible test cases for all parts of a GUI, because the number of event sequences and input value permutations could be extremely large or even infinite making exhaustive testing impossible. Thus, how to select an effective and efficient subset of test cases for GUI testing is a crucial concern. A systematic testing strategy is needed, which is not only easy to apply, but also scalable in the sense of increasing the test complexity in a series of steps and, accordingly, the test coverage and completeness of the test process.

To date, research on GUI testing has not proven very fruitful. Some notable exceptions include the following work. Originally, in White [1996] researchers applied the concept of Latin Squares for GUI test case generation to reduce the number of test cases substantially. However, their assumption to test only pairs of GUI objects seems to be too limiting, as more complex GUI interactions also cause many failures.

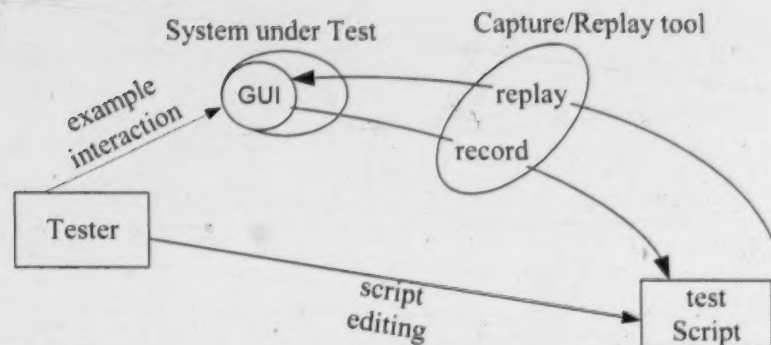


Figure 1.1 A Standard Commercial Capture/replay Tool

Subsequently in White [2000], they introduced the concept of Complete Interaction Sequences (CIS), which, to some extent, helps partition the entire GUI system to sets of independent tasks. Doing this reduces the complexity of GUI testing. Their steps to process GUI testing include:

First, responsibilities are identified using all the available information sources concerning the GUI system. A responsibility includes some GUI activities applied on one or more GUI objects to produce an observable effect on the surrounding environment of the GUI.

Second, for each identified responsibility, the corresponding CIS is defined. A CIS consists of the sequence of relative GUI objects and selections that will invoke the responsibility.

Third, for each CIS, a finite state machine (FSM) model is constructed. In order to simplify the testing, some state reduction algorithms are applied to produce a reduced FSM for each original FSM.

Finally, tests are generated for each component according to its corresponding reduced FSM, which is represented as a directed graph. Each test takes the form of a directed path through a component, starting with a specific input to the component, and terminating with a specific output from that component.

White [2001] further investigated the use of memory tools to detect missing effects and the interactions between different CISs. Belli extended this method by

complementing Fault Complete Interaction Sequences to test GUIs in a holistic view (Belli [2001]).

Another notable work is the goal driven approach proposed in Memon [2000b]. Their team suggested using AI plan generation techniques to generate testing information. A GUI represents a typical planning problem: given a set of operators, an initial state and a goal state, a planning system, which is implemented in Planning Assisted Tester for graphical user interface Systems (PATHS), produces a sequence of operators that transforms the initial state to the goal state. These operators are actually a set of test cases for GUI testing. The additional benefit of this method is its ability to automate the construction of a testing oracle (Memon [2000a]). Their test case generation process includes two phases: the *setup* phase and *plan-generation* phase.

In the setup phase, first PATHS creates an abstract model of a GUI and uses that to produce a list of operators representing GUI events. Then test designers use his or her knowledge of the GUI to define the preconditions and effects of the operators. The preconditions represent all the conditions that must hold for the event represented by the operator to occur, and the effects represent the resulting changes to the environment.

In the plan-generation phase, the test designer describes a task first. The task is defined by a set of initial states and goal states. Once the task has been specified, PATHS automatically generates a set of distinct test cases that achieve the goal. The test designer can iterate through the plan-generation phase a number of times, defining different scenarios and generating more test cases. It is important to generate alternative plans for each specified task, since these correspond to alternative ways in which a user might interact with the GUI.

Additionally, Memon [2001] defines a set of Coverage Criteria using event sequences to specify a measure of test adequacy for GUI software. However, some points about the coverage criteria require more clarification. In the provided study case, there are many unfeasible event sequences in the total event sequences. In the case study, 21,659 event sequences were generated up to the length of 3. 4,189 (or 19.3%) of them were unfeasible. This percentage is likely to significantly vary between systems.

Hence, using the number of total event sequences to evaluate the coverage of a test suite can be rather inaccurate.

Specification-based testing has become popular recently. Dwyer applies symbolic model checking to the specification of a graphical user interface system and its implementations (Dwyer [1997]). But the result of this method completely relies on the quality of the requirement specification of the GUI software.

There have been a number of papers, although not specifically focused on research on the GUI testing, which have provided some interesting insights into this problem. L. The. [1992] discusses defects in an environment caused by a GUI system, for example, memory leaks and garbage collection, as well as stress testing for GUIs. Kepple [1994] describes "The Black Art of GUI Testing", suggesting that tools identify the location of an object displayed on the screen, eliminating synchronization and timing problems between GUI responses and a tool, rather than just blindly relying upon capture-replay to solve all testing problems. Remy [1998] also mentions the usefulness of state diagrams for GUI testing, and he recommends that a GUI be divided into appropriate subsets of objects for testing. Gerrard [1997] attempts to formulate a GUI test strategy to detect errors and discusses how to use tools to assist in this process. He supposes organizing different test types into a series of test stages in order to cover all error types.

The methodology proposed in this thesis can be viewed as an extension of the early research, simple and easy to apply in practice. It targets the special difficulties of GUI software testing. An empirical investigation demonstrates its effectiveness in finding defects and its reasonable cost of application. In the following sections, we will discuss the details of our approach.

2. A TWO TIERED GUI TESTING STRATEGY

2.1. Overview

The strategy proposed in this paper is to methodically apply testing to GUI systems on two abstract tiers – the component tier and the system tier. To avoid confusion, we have to point out that this approach sits at the system testing level, in which all testing objectives relevant to specific requirements should be included during the testing. Software system testing is mainly based on black-box methods. Hence, we have avoided code-based or structural testing. This two-tiered GUI testing is the extension of the finite state machine based GUI testing method and covers many details that are crucial for GUI testing but often overlooked by others. Some techniques used in this method are built upon previous work. The testing is broken into tiers and on different tiers models are applied for different testing objectives. The following is an overview of our approach.

First, we subdivide the GUI software under testing into components. There are several ways to apply the decomposition. Memon [2001] defines a GUI component in terms of modal windows. "A modal window is a GUI window that, once invoked, monopolizes the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly terminated." So a GUI component is composed of a modal window and one or more modeless windows. In White [2000], the component of a GUI is identified according to the "responsibility" of the software, which means that GUI activities are applied on one or more GUI objects to produce an observable effect on the surrounding environment of the GUI such as a memory change, a change in the behavior of a peripheral device, or a change in the underlying software or application software. For each responsibility, a complete interaction sequence is defined correspondingly.

In the current approach, the concept of "responsibility" is still used and we will introduce the process to identify the responsibilities of a large GUI system in Section 2.4. For each responsibility, a corresponding component is defined, which is an aggregation of all the GUI objects that are possibly involved in this responsibility. This method stems from the consideration that some modal dialogs can be involved in

many features of a GUI. There are many events that can trigger the same modal dialog. Furthermore, in many situations, whether an event invokes a modal dialog or not depends on the environmental conditions at that moment when the event is triggered. Grouping the GUI objects according to modal dialogs can make the interfaces between various components complex and obscure. For example, in the Microsoft Notepad application, the "Save" dialog is a modal dialog, which collects information from users about where to save the text contents. It can appear when users try to create a new file, open a file, save a file, or even exit the application. Several menu events can invoke this dialog directly or indirectly, depending on whether the text area has unsaved contents or not. On the other hand, grouping GUI objects in terms of the functions of GUI software is more natural and similar to how users interact with GUIs. We will discuss the process of how to decompose a GUI into components in Section 3, component testing.

A component can have one or more functional granules, which are usually composed of many GUI objects. These objects gather information from users and then implement certain GUI functions before giving the information back to users. They include the GUI object to launch the function, associated dialogs or windows, warning messages, etc. A flow graph is constructed for each of these components. As long as a GUI object is likely to be reached during the application of the function in a certain situation, it should be present on the flow graph. As in traditional branch testing, a flow chart is a convenient representation of a program. Here, our flow graph is a network of nodes and edges to represent the state transfer between GUI objects. A node in a flow graph represents a GUI object or a decision-making process in terms of the environmental conditions instead of a block of code. Edges on the flow graph represent users' GUI activities on GUI objects that cause the invocation of an object and/or the closing of another. We will give detailed examples of flow graphs in the next chapter. A lot of analytical work on the GUI's specification, design, or even implementation should be done to build a flow graph. Each graph has an entry node and an exit node. We borrow the concept of a path from graph theory - a path traverses from an entry node to an exit node through the GUI objects belonging to a function. Actually, each path is a set of original state parameters, a sequence of events, and the corresponding outputs. So given all paths of the flow graphs, test cases for the GUI component tier are produced.

As discussed in the previous section, the interplay between components is the other concern of GUI testing. Testing the interactions between all of the components is looked at as testing on the abstraction level of the entire GUI system. A finite state machine (FSM) contains a finite number of states and produces outputs on state transitions after receiving inputs. It has been widely used to model the behaviors of systems in many areas and is used as the model for our system's tier testing. We discuss the details of the FSM in Chapter 4, in which a state transition diagram represents the FSM model of the GUI software under test. All the identified responsibility components will be inputs to the FSM. All edges between the inputs imply transitions of a GUI from one state to another. Meanwhile, users navigate through the functions implemented by the GUI system. This model corresponds to the main window of a GUI application, which organizes all of the components into a hierarchy and from which all GUI components can be triggered by end users. On this tier, the testing objective is to evaluate the operating sequences of the components that are likely to occur in practice. Once the state transition diagram has been constructed, we can identify entire sets of event sequences as edges of the FSM. Corresponding test cases are produced to satisfy edge coverage, which we define at this tier.

Before discussing the details of the proposed approach, it is helpful to introduce a very important design pattern for most GUI-based programs – the Model-View-Control (MVC) pattern that decouples the graphical user interface of an application from the underlying code. As implied by the name, this pattern breaks up an application into three logical units: the model, the view, and the controller. The model is a set of objects that represent the business data of the application; the view is a particular way of presenting information to users; the controller is the layer in an application that sits between the view and the model. When a user presses a button, some code might run and perform some operations on the data, perhaps updating some of the output displays as a result. The approach proposed in this thesis only focuses on the tier of the view to verify the behavior of it under varied inputs. In general, we do not care how to implement the business logics and how to process the data between the view and the model.

2.2. Why We Need a Two-tiered Approach

Effective testing cannot be achieved without using abstraction to conquer the astronomical complexity of typical software systems. As discussed in Chapter 1, GUIs have so many special characteristics that cause creating models for their testing involving many special considerations. GUI software differs from the underlying application code in its level of abstraction. The mapping between GUI appearance or events and the underlying code is not simple or straightforward. So the traditional methods of test case generation and testing adequacy evaluation cannot be applied to GUI testing paradigms directly. However, this does not mean that all the traditional testing techniques and analytical methodologies are inappropriate for GUI testing. Actually, many are still crucial, but have often been overlooked by others in this field.

The requirements of modeling a GUI system and other software systems are very similar. Binder [2000] gives much useful advice on creating models for Object-Oriented system testing, which are applicable for modeling many other systems. The requirements of a testable model for a software system are:

- It is a complete and accurate reflection of the kind of implementation to be tested. The model must represent all features to be exercised.
- It abstracts detail that would make the cost of testing prohibitive.
- It preserves detail that is essential for revealing faults and demonstrating conformance, etc.

The objective of creating a single model of a GUI, which can represent all features within a system and abstract out all details of the system without losing essential information for revealing faults, is actually difficult to accomplish. Especially for a large and complex software system, it is impossible to detect various types of errors based on a single testing model. It is wise to separate testing concerns for the system and conquer them respectively. Each type of model has its advantages and weakness. For different testing concerns, appropriate testing models are applied to achieve the optimum result for the entire system. This is the key point of the proposed systematical testing strategy that methodically applies a two-tiered approach to GUI testing. On different tiers, different testing models are applied to cover different

aspects of GUIs. Our heuristics come from analyzing the characteristics of GUIs and previous work by others in this field.

When we look back at the characteristics of GUIs discussed in Section 1.2, the most challenging issue for GUI testing is its large input space and many intertwined event-handling processes behind the GUI objects, which result in a huge number of states of a GUI requiring to be tested. The inputs to a GUI are more complex than the inputs to the traditional form-based interface, in which users' activities are limited in one form at a time. In a GUI, users are given more freedom to interact with the system than any other previous types of interfaces. GUIs put more attention on the needs of human beings than the other mechanisms. Human factors play key roles in designing, developing and testing GUIs. Users' inputs to a GUI not only cause some data or selections to be passed back to the underlying software but also involve different sequences of activities conducted on GUI objects. Because each activity has a corresponding event handling process, different sequences of activities mean different sequences of executions of their processes and may cause different outputs of a GUI or lead a GUI to different states. Unless a GUI physically constrains users to special sequences of activities, users can and will use a "feature" within a GUI by many sequences of activities. Numerous states of a GUI cause the testing to be time consuming and labor intensive. Even with record-playback tools, GUI testing is not easy work. Therefore, testers have to design effective models to represent the system and meanwhile ensure that appropriate assumptions of user behaviors are made. This is the crucial point of why the approach in this thesis applies GUI testing on two different abstract levels with two types of models. The following sections show the detailed reasons and associated techniques on both levels.

As discussed in the previous sections, the desire to design a single model satisfying all these requirements is impractical. Intuitively, any scalable representation must decompose a GUI into manageable parts. There are several reasons for this consideration:

First, today GUI software is large and provides many features or functions. Even during the early requirement specification or design phase, there is a trend to divide all the functions or tasks into groups. Although this concept of the group is not

necessarily the same as the one in this thesis, it is still wise to apply testing by subdividing the software.

Second, by nature GUIs are hierarchical because of the syntax and construction mechanism within various programming languages. The top-level frame comprising the application window may contain sub-frames, which in turn contain still more frames or controls. These controls can be visualized as a tree structure with each control having a single parent and a number of children. This hierarchy may be exploited to identify groups of GUI objects and associated events that can be tested in isolation. So decomposition of the whole system into simpler independent subsystems is necessary and reasonable.

In this thesis, a GUI is divided into components, each of which forms a unit of testing. As in performing traditional testing, a unit of testing may be a basic block, a procedure, an object, or a class, all of which can be tested independently. In our GUI testing approach, each component can complete a certain responsibility of a GUI. From the point view of the whole system, a component interfaces with the other parts of the software. It can be invoked and terminated by other components when needed. The approach in this thesis will be to apply GUI testing in two stages: component tier testing and system tier testing. On both tiers, different mechanisms are used to generate test cases based on their models respectively to conquer different concerns of GUI testing.

Software test design is a process of identifying a set of inputs executed to produce expected outputs that satisfy certain testing coverage criteria. For GUI systems, test inputs usually represent sequences of GUI object activities that operate interactively with the objects. These activities on GUI objects can include mouse clicks and/or keyboard strokes stimulated by a human user. After the activities, events and data are passed back to the software application under test. Some objects, such as buttons, have very simple events that they can pass; other controls pass not only numerous events but also data as well; for example, a list box that returns a string that a user highlights. Furthermore, it is common for several GUI objects to be used to collect related data for an action. The tester has to discover the relationships among the input combinations of these GUI objects. On the other hand, although most GUI objects

provide selections for users to pick from, in many situations, GUIs do not limit users to input illegal or erroneous inputs. Hence, GUIs have to filter all the illegal and erroneous inputs to make sure that the software accepts inputs that it can handle and give users proper warning or error messages for bad inputs. So input space analysis for GUI objects and, further, for the whole GUI system is still very important. Inputs and input combination validation is the most likely reason to cause software to fail. An interesting instance showing its importance occurs in Microsoft Word 2000. In the "Insert" menu, select the "Index and Tables" tab, change the "Column" field to five and then press the enter key. Since the maximum boundary is four, this action will cause the application to correctly display an error message. However, the error message appears twice in a row for no reason. Although this is not a devastating error, it is very annoying (Whittaker [2002]). Figure 2.1 illustrates this bug. These bugs are visible, if users see many, they are less likely to buy or trust the program. Unfortunately, GUI bugs are often considered minor and not investigated. This is risky; they may be symptoms of more serious underlying errors.

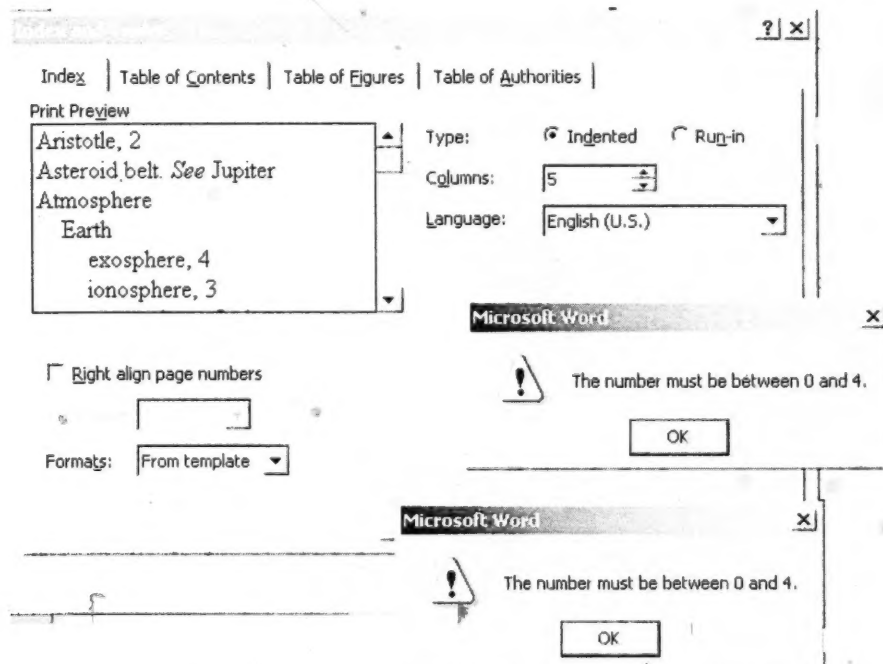


Figure 2.1 A Bug in Microsoft Word 2000

The testing objective on the component tier is to verify whether each component completes its corresponding responsibility correctly with various input space divisions. The testing model on this tier is a flow graph combining with state based models of involved GUI objects. Some well-known heuristics for test case selection including equivalence classes, boundary value analysis, and special values testing (Myers [1979]) still play a key role here. The flow graph model facilitates the application of these testing techniques.

When we build a model for GUI testing, the other crucial concern is the order in which GUI objects are used. The most well known character of GUI applications is their event driven nature. Inputs to a GUI are actually sequences of events. Different event sequences cause different paths to be followed within the software. Compared with previous command-line interfaces, with GUIs, users are given more freedom to navigate through software applications. The number of potential paths from feature to feature within a GUI are so high that the scope for programmers to make errors is dramatically increased. Furthermore, generally during software design and development, certain execution orders through the software by users are often decided. Users are expected to use certain objects before other objects. So developers write code under the assumption that users will play by these rules. Testers must ensure that these assumptions about how users will use the software are in compliance with the specification. The model for a GUI should have the capability to represent the key event sequences that can lead to different paths within the GUI. At the same time, the model should not be overwhelmed by a huge number of event sequence permutations, making the number of test cases generated too large or unlimited to apply in practice.

On the system tier, the description of the components is integrated to build up a viewpoint of the entire system. Tests on this tier, try to interrogate the interactions between components. A finite state machine model is created to represent the system on this tier.

As explained in the previous section, when test inputs are designed for GUIs, these important things should be taken into account: what the event is, the data input space to a GUI object, and what are the sequence of activities upon the GUI object. Ignoring

any of these aspects is unacceptable and can lead to bugs being missed during the testing process. Unfortunately, early GUI testing research paid attention only to the event sequences; input space analysis has been neglected. In many circumstances, input space analysis plays an essential role in selecting test cases. Actually, sometimes different orders of events on GUI objects are not very meaningful for testing. By contrast, input values falling in alternative ranges of a GUI object can affect what the GUI's next appearances are. Any value in the same range leads to the same or similar corresponding effect for a GUI. The data samples from different input ranges to a GUI will cause end users to go through the application by different paths. The model proposed in this thesis tries to cover both sides of the problem.

2.3 Introduction to the Example GUI System Used in the Research

We applied two experiments to investigate the effectiveness of the proposed approach. In the following chapters, we will use the software system used in our first experiment to explain how to apply the two-tiered approach. Hence, this section gives a brief description of the first system. The details of the two experiments will be discussed in Chapter 6. The software system used in the first experiment is a clone of a Notepad system, which is a typical GUI application and in fact, closely mimics Microsoft's Notepad. It is a basic text editor, which you can use to create simple documents. The most common use for Notepad is to view or edit text (.txt) files. Text will appear with no formatting (cool fonts, bold, etc). We implemented it using Java. The software defines 9 classes consisting of about 1400 lines code of Java. It includes a main window, four menus (we didn't consider the help menu in our investigation.), five dialogs and a number of associated message boxes. In the following sections of this thesis we will provide many examples to show how to apply our approach, for convenience we present some snapshots of Notepad in advance. Figure 2.2 shows the main window of Notepad. Later we will show more snapshots on other parts of the Notepad application when required.

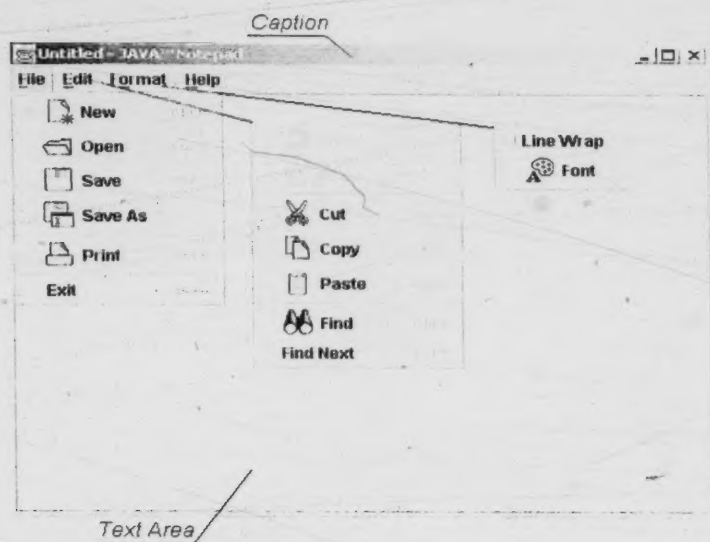


Figure 2.2 The GUI of Notepad

2.4 Responsibility Identification and Example

The first step to apply the testing approach in this thesis is to identify all responsibilities for a GUI system. It is a process to divide a large, complex GUI into independent components, which can be tested in isolation. In other words, it is the process for setting the boundary to apply the different techniques. Components will be constructed according to their corresponding responsibilities. Tests on the component tier, will deal with the validity inside the component. The testing on the system tier, conducts test cases to verify the relationship between the components.

The concept of the responsibility first appears in White [2000], where a responsibility is defined as GUI activities applied on one or more GUI objects to produce an observable effect on the surrounding environment of the GUI. To some extent, this definition is ambiguous. The "observable effect" can be very small or very big, so the corresponding responsibilities can be as simple as involving only a couple of activities or as complex as involving several windows. There is no clear idea about the size of a responsibility and how to identify it. In this thesis, we still use the concept of responsibilities but we provide a clearer definition and practical guidelines to identify them of a GUI through the lifecycle of software.

2.4.1 The Responsibility Definition and the Identification Criteria

In our method, a responsibility is defined as a group of activities performed by users on one or more relatively isolated GUI objects to complete a task. Here, "relatively isolated" means that each responsibility can be regarded as a process comprising of one or multiple steps, which once invoked will hold users' focus until it is finished or aborted. The logical relationships among the steps are exactly defined. Each responsibility is a functional granule of a GUI.

In the above definition, two characteristics of a responsibility should be adopted as the criteria for identifying a responsibility.

- A responsibility should complete a task defined in the system.
- A responsibility should have a group of relatively isolated GUI objects and corresponding operations.

2.4.1.1 Criterion 1

In this definition, "task" refers to the functions that are specified in the software specification. Generally, to invoke and apply the task requires input from users. A task directly or indirectly produces observable effects on the GUI and its environment. For instance, most software has dialogs developed for changing various environmental settings, which include multiple data entries. When users conduct the setting function, no matter whether the data entries change the GUI appearance, certain parameters from the application environment must have been changed as required.

Generally, the software requirement specification defines all the functions that should be implemented in the system. These functions can be very complex and include multiple parallel workflows. Subsequently in the lifecycle, further analysis will be conducted to divide the functions into multiple sub-functions and sub-functions can be represented by several user activities. When the relationships among these activities are fixed, we define the process made up by these activities as a responsibility and the collection of the corresponding GUI objects as a component. Hence, the identification process begins from the top-level functions going down to

low-level functions. And the working flow can be represented with a flow graph. This process has two benefits: on one side, the produced responsibilities are not too small, and trivial, to cause the model of the entire GUI to become extremely complex; on the other side, the involved GUI objects of a responsibility can be isolated for testing from other objects in the GUI. As long as the responsibility is invoked, a user's focus is confined to certain GUI objects. The possible sequence of these GUI objects is countable. The details about how to identify the responsibilities will be discussed further in Section 2.4.2.

2.4.1.2 Criterion 2

A responsibility involves a group of relatively isolated GUI objects and activities. In general, users can randomly use any functions as they want. However, when they start a responsibility, their activities will only focus on the GUI objects associated with the responsibility. The sequences of these GUI objects are clearly defined according to the user's activities and the environmental conditions. Actually, the environmental conditions are the effects induced by the previous user activities in the GUI. Such conditions are defined by the variables of the GUI; different values of these variables define different states of the GUI. In system testing, the corresponding components of all of the responsibilities are used to represent the finite state machine of the entire GUI. A set of values of these environmental variables is the guard to the related state of a GUI. Only when the guard is satisfied, it is possible to transit to the corresponding state.

Responsibilities represent the basic function atoms in a GUI. As long as users invoke one, they cannot access all of the other responsibilities until it is finished. For example, a responsibility of Notepad is the "Create a new text file", which involves GUI objects: File -> New menu item, Confirm save message box (Figure 3.9), Save a file dialog (Figure 3.10), and the text area. In defined situations, users have to follow certain routes to interact with these GUI objects. Testing inside a responsibility is to verify whether the responsibility is processed correctly in varied environments and with different inputs from the users. A flow graph is a good tool to represent this structure and facilitate test generation. The flow graph of "Create a new text file" is shown in Figure 3.11. The details of this example will be discussed in Chapter 3.

However, the isolation of responsibilities is relative. Not all the responsibilities strictly confine users' activities inside themselves. We still use the "Create a new test file" responsibility as the example. The activities to invoke it actually include two steps: clicking the File menu and then clicking the New menu item. When clicking the File menu, there are still many other menu items available for users that are not objects in this responsibility. In this situation, we define the two steps as one atom, which contributes to apply the responsibility.

Another concern is about the GUI objects. Some GUI objects can include more sub-objects. As discussed in the example of "Create a new text file", the Save a file dialog has buttons, lists, a combo box and so on. These objects are on the same level; no matter what sequences users apply, the whole set of variables to the objects will not be passed down to the under code until users close the dialog. Hence, those objects are treated as one GUI object. In general, during responsibility identification, we only consider GUI objects such as menu items, dialogs or sub-windows, and controls on tool bars, as the analysis objects.

2.4.1.3 More Discussion on Responsibility

Only when the two criteria are simultaneously satisfied, can a set of GUI operations be accepted as an effective responsibility. We will discuss how to identify responsibilities along with a development process in the following section. In this section, some commonly confusing topics about responsibilities will be demonstrated

First, in many GUIs, a specific function can be activated by several activities. For example, 'Copy' can be invoked through menu, tool bar, short-cut keys by keyboard and short-cut menu by right clicking the mouse. So the question arises: whether is it necessary to test the function multiple times? Our answer is that these operations that fulfill the same task are independent activities. The responsibility is just invoked by these different user activities, each of which is treated as an edge in the flow graph. The component tier testing covers all the activities inside a responsibility. This does not conflict with the two characteristics of a responsibility.

Second, in the system tier, a finite state machine is used to represent the entire GUI system. The sequences of activities that implement the responsibilities are the input event sequences to the finite state machine (The details of how to construct the model

on the system tier will be discussed in Chapter 4). Hence, the number of identified responsibilities will significantly affect the scale of the model. For a complex GUI system, the number of its responsibilities may be large. It causes the structure of the finite state model to become very complicated and too expensive to be tested. A way to deal with this problem is to group certain responsibilities together as a compound state of the finite state model, which can be represented as a sub-finite state machine as well. In several situations, it is reasonable to group responsibilities. In one case, when a function has multiple sub-functions identified that do not have interrelationships among them and have similar relationships with other sub-functions of the GUI (It means that users do not have to use them before or after certain responsibilities), we consider to group these sub-functions together.

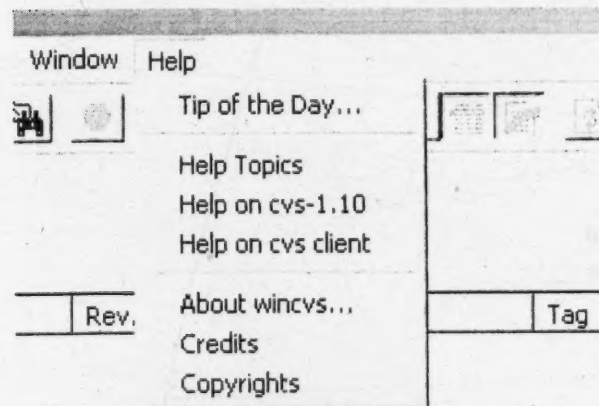


Figure 2.2 A Typical HELP Menu

Figure 2.2 shows the HELP menu of WinCvs software. There are seven sub-functions under the HELP function identified and all of them have no interrelationship with other sub-functions. Users can just invoke these functions whenever they want. We assume that each of these sub-functions is a responsibility; and there are other 30 more responsibilities of the system. If these responsibilities are not grouped together, the number of edges connecting them together is approximately 250. When these sub-functions are grouped together, the number of edges among them is about 100. In general, this compromise does not impact the testing effectiveness and saves a lot of work.

The other case where this simplification has to be applied is for complex sub-windows. Usually, a GUI has a main window where users can invoke all functions of the application. However, for some complex functions or applications from third parties, sub-windows are used to organize related GUI objects to fulfill the functions. The responsibilities implemented in the sub-windows can be grouped together to form a sub-finite state machine. With this simplification, a GUI can be represented in hierarchical levels. In WinCvs, when users want to check the revision history of a file in the work area, a sub-window is launched to show the graph. The sub-window includes many responsibilities associated with this function and has different menus from the main window, which are shown in Figure 2.3. All responsibilities under this function should be treated as a group when constructing the finite state machine on the system tier.

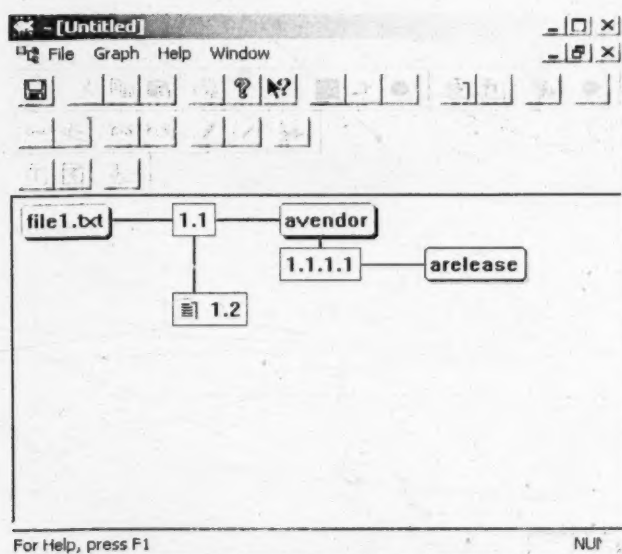


Figure 2.3 A Sub-window of WinCvs

2.4.2 Identify Responsibilities

The process to identify the responsibilities of a GUI begins with gathering and exploring a variety of reference materials for the system. The possible sources include the requirement specification, the design document, the user manual, and even experience when interacting with the system. Actually, these documents are necessary for various types of testing at many stages throughout the life cycle, and are not only

for the approach proposed in this thesis. According to our experience, during the information gathering process, many defects or ambiguities within the GUI can be found. The more testers understand the system under test, the more exactly the model that they can build to represent the system. However, in practical projects, testing is always a process that has to balance its cost and effectiveness. And in some cases, several stages of the software development may be skipped and not all of the previously mentioned resources will be available. Testers have to work on the documents that they have. Developers have to produce the correct product against the requirements and testers try to uncover the gap between the definition and implementation with the available resources. In the following part of this section, we try to provide some guidelines and examples to describe how to explore the available information to identify the responsibilities.

According to the definition, responsibilities are the functional granules on the bottom level, which can be grouped to build the top level functions and the entire system. However, the process to identify the responsibilities starts from the top-level functions and goes to the bottom level through the lifecycle of the software.

Literally, software testing aims to verify whether the software meets the requirement of customers that are defined in the Software Requirement Specification (SRS). Testing is a parallel effort that starts with the original requirement stage and evolves with the progress of design and developing work. The typical software development process is shown in Figure 2.5. Conceptual design, detailed design and programming stages add more information to the testing step by step.

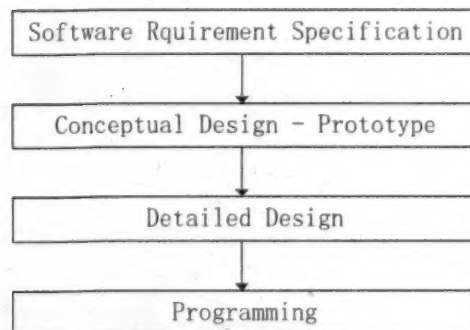


Figure 2.5 Developing Process

2.4.2.1 Identify Responsibilities with Software Requirement Specification

The most important document for testing is the Software Requirement Specification (SRS). For testers, "no specification means no testing." at least in a black-box sense. It is a fundamental document, which defines what the system is required to do and what constraints are placed upon the implementation. It plays a variety of roles during software development: the basis for the customer and the supplier to make the contract, the basis for the manager to make the schedule, the basis for the designer to develop, and the basis for testers to make a test and verification plan. According to IEEE standard 830-1998, the SRS of a software system should comprise specifications for its functionalities, external interfaces, performance, attributes, design constraints, and non-functional requirements.

The identification of responsibilities begins from the analysis of the SRS. Among all of the information provided by the requirements, the section on "product functions" is valuable for testers to identify the responsibilities of the system. The major functions and their sub-functions that the software should perform are described in this part. For example, a SRS for an accounting program may use this part to address customer account maintenance, customer statement, invoice preparation and so on. For each function, functional requirements should describe the processes of the function, the input and the corresponding output. Hence, we know which part should be done by the system and which part should be done by users. Another example is shown in Figure 2.6, which is a fragment of the SRS of a video rental system. From a SRS, testers can make a list of functions and the processes to implement these functions. This function list is the starting point for identifying responsibilities. However, the SRS does not describe many of the details about the GUI but gives testers an outline of the features and functions within the system. At this moment, what exactly the GUI should look like is unknown. For each function, what kind of GUI objects will be used to implement it has not decided. Without these essential details of a GUI, to generate accurate responsibilities is unrealistic.

3.1.1 Requirements for renting a tape
Functional Requirement 4

- *Description*
The customer has his ABC card with him. The account number of a customer is read with the bar code reader from the clerk to retrieve rental transaction record.
- *Input*
The account number is entered via bar code from account card.
- *Processing*
Searching rental transaction record.
- *Output*
Display rental transaction record.

Figure 2.6 A Fragment of the SRS

Although, the function list is only an approximate list of responsibilities and some of them need to be divided into finer ones, it provides the starting point for GUI testing. Along with progress development, the responsibilities will be refined to be more and more accurate and more rational for the testing model construction. Therefore, the testing models finally generated on the base of them can correctly represent the GUIs. In summary, responsibility identification evolves gradually in the testing process as well as the development process. For our Notepad application, the function list is as followed:

- Users shall be able to open and see the contents of a text file.
- Users shall be able to apply edit functions on the text.
- Users shall be able to save the text to file-system.
- Users shall be able to print out the text.

In the next stage of software development, the design stage, designers figure out how to provide the planned capabilities of the software system. Many valuable resources to identify the responsibilities of a GUI can be achieved in this stage.

2.4.2.2 Identifying Responsibilities within the Design Stage

Software design is the first step of implementation of a SRS. The design aims to map the various requirements to technically feasible solutions and to reason about the

correctness of the approaches before implementing the solutions. Requirements may be specified using several views such as behavioral (use cases), informational (object models), and "nonfunctional" (ad hoc and other methods). The design maps these various inputs to software entities such as objects. The result of the design phase is a specification of how to build the application or system and the technical constraints on the implementation. A functional requirement may be divided into sub-functions and mapped to several related components, which when executed together must satisfy the requirement.

For GUI testing, most testers might prefer to identify responsibilities when the design is finished because the output of the design is more observable and detailed than the SRS. There are two types of designs: external and internal design. The external design describes the product from the user's point of view. The internal design describes the internal working of the product (Kaner, Falk & Hung [1999] and Albin [2003]). Usually, the external design explores more information about the interface, such as, "how are the functions related to each other, sequential or concurrent?"; "what data is required to perform each task and how is the data to be transformed by the system?". It produces a complete description of the user interface – the User Interface Specification, which defines the appearance of the GUI and how users interact with the screens and does not address the software implementation. It is also an important document to verify the behavior of a GUI during testing. The functions listed on the SRS are interpreted on the screens. It describes in detail exactly how to perform each function, every sub-task and step, and how to use the system to perform the functions. Generally, a product manager, a graphic designer and programmers work together on this stage to create the screen-to-screen design of the software. A good design of the user interface involves numerous technical considerations, such as how to organize information, how to select proper type of windows and control, and how to decide the sequence of screens, etc. ([Galitz 2002] provides guidelines and principles for this process.) In this stage, the major functions are divided into finer sub-functions. For GUI testing, we are only concerned with the functions that require users' interaction through the GUIs. For each of these functions, the associated screens are decided: what kinds of windows or dialogs are used, what types of GUI controls are designed to get users' inputs and what the corresponding outputs are. By now, the workflow for each function is created.

3.2.1.4.2 BRIDGE

The Bridge is the entire model. There are some actions that can be carried out on the entire bridge as follows

3.2.1.4.2.1 Move Bridge

This command translates the entire bridge along the alignment a given distance, or moves the location of the first pier to the desired station.

Move Bridge

This dialog is used to move the entire bridge along the alignment.

☒ Translate along alignment ft (negative values move bridge down-station)

☐ Move Pier 1 to Station

OK Cancel Help

Figure 43 - Move Bridge

3.2.1.4.2.2 Edit Span Lengths

This dialog allows editing of span lengths directly in lieu of pier stationing.

Edit Span Lengths

Span	Length (ft)
1	50.2
2	42
3	51.4

Span Length Change Moves:

- ☒ Left piers
☐ Right piers

Ok Cancel Help

Figure 44 - Edit Span Lengths

Figure 2.7 A Fragment of a User Interface Specification

In the user interface specification, the prototype of each screen is included to show how the GUI should look. It is a valuable vehicle for exploration, communication, and evaluation on the interface design. A prototype intends to obtain user input, and to provide feedback to the designers especially for evaluating the functionality and usability of the interface. There are several ways to create prototypes from simple

hand sketches to more complex ones using prototype-oriented languages. Such prototypes need not be perfect and complete, but visualizes how the software will look and work. A prototype provides adequate information of the content and structure of individual screens, the overall orders of the screens and windows, which can help testers to organize GUI objects for the testing task.

Figure 2.7 shows a fragment of a user interface specification and some prototypes of the system. In some cases, a user interface specification may include a table of events and responses of each function, which can help us understand the process of the function. For instance, in the Notepad, the event and response table of the function "Open an existing file" is shown below:

Table 2.1 An event and response table example

Event	Response
Single click on File menu	File pull down menu list appears
Single click on Open menu item	Open dialog box appears.
Etc...	

The design stage keeps going on until it is detailed enough to be implemented in software. By now, the testers know the details about each function, how it is divided to sub-functions from the GUI's point of view and how users apply them through the GUI step by step. We analyze each task to see whether it fully complies with the fore-mentioned two criteria. If it does, it will be a responsibility; if it does not, the function will be divided into sub-tasks until the finest atoms are reached. The corresponding GUI objects of the finest atoms can be a dialog, a message box, a button on the toolbar, or a menu item. The activities by users on the related GUI objects form a sequence of events to complete a task in GUIs.

Table 2.2 Part of the Notepad Function List

- | |
|---|
| <ul style="list-style-type: none">1. Users shall open and see the contents of a text file.<ul style="list-style-type: none">1.1. Open an existing text file.<ul style="list-style-type: none">1.1.1 Check whether the current text area needs to be saved.1.1.2 Get storage path and file name1.1.3 Load the contents of the file in text area.1.2. Create a new text file.<ul style="list-style-type: none">1.2.1 Check whether the current text area needs to be saved1.2.2 Set the text area blank... |
|---|

Table 2.2 shows part of the detail function list of the Notepad Application. We can identify the responsibilities of it according to the list. Thirteen responsibilities are finally identified as following:

- 1. Open an exiting text file.
- 2. Create a new text file.
- Edit a text file.
- 3. Redo a canceled action
- 4. Undo the last action
- 5. Cut
- 6. Copy
- 7. Paste

8. Wrap line
9. Find a word in the text.
10. Set the attribute of the text's font.
11. Save a text file.
12. Print out a text file.
13. Exit the application.

We have to pay attention to two potential risks. Some software development projects might skip conceptual design or even whole design stage but program the source code directly according to the SRS. Moreover, the design of the GUI could deviate from the SRS for a variety of reasons.

For the first scenario, responsibility identification can be delayed to the detailed design phase or the end of programming. To deal with the second risk, we have to check the identified activities in light of the SRS to guarantee the consistency. If responsibilities have been identified before the design, they need to be refined based on the clearer logic among or inside them after the design has been finalized. Otherwise, the identification should be conducted here and the consistency between SRS and design should be checked.

This responsibility list is the first draft for GUI testing. In the later stages of the software life cycle, modifications to the functions or sub-functions are not completely avoidable. So, after the implementation stage, before applying testing on GUIs, the responsibility list will be reviewed for any change from the first draft.

2.4.2.3 Identify Responsibilities within Implementation Stage

Basically, GUI testing does not care about the details of the implementation. Unless the system is pretty simple; the design is not detailed enough or the design stage has been omitted, there is no rationale for identifying responsibilities from the implementation.

If this scenario does happen, it is a good strategy to leave the testing job to the end of the programming effort and after the unit test has been finished. Testers can get more

information from the direct interaction with the GUI. According to the functions listed in the SRS, testers have to explore all the associated GUI objects with each function, all selections of the objects, and the input data type and range, and so on, to check whether the functions can be divided into sub-tasks. Then responsibilities are identified on the base of this work. Actually, this strategy sounds more like Beta testing in which the testers (some dedicated customers) test the software by using the system to see whether it complies with the requirement specification. However, the apparent difference between this strategy and the Beta testing resides in the testing methodology. Beta testing is conducted randomly by the end users but the strategy discussed in this section is systematic and follows strict rules. In the testing methodology proposed in this thesis, testing is applied on the component tier and system tier. A component is a collection of GUI objects of a responsibility. Since we cannot identify responsibilities directly from the SRS, we have to interact with the software itself to identify the responsibilities based on the two criteria discussed above.

Generally, the functions of software will be mapped into the menu system of its GUI. Each function can be invoked from the menu system. At first, we can assume each of the menu items as a responsibility and then explore the process of them. There are several situations:

- If the menu item invokes a task directly without any other GUI object, we can define it as a responsibility.
- If the menu item invokes a modal dialog (before closing the dialog, users' focuses are constrained inside the modal dialog and its associated modeless dialog and message boxes), we define a responsibility for the entire group of these GUI objects.
- If the menu item invokes a modeless dialog, users can use it whenever they want. The function of this dialog is a responsibility.
- If the menu item invokes a sub-window, the process to identify responsibilities is recursive.

In summary, the SRS is the bottom line that the final software should satisfy. When the identified responsibilities show different behavior from the definition within the SRS, they have to be corrected according to the SRS.

Another information source that is helpful with responsibility identification but has not been mentioned in the previous sections is the user manual. They describe what the software is designed to accomplish, and how to perform the tasks necessary to accomplish the software functions:

In a cycle of software maintenance, the work begins with SRS amendment. When the new SRS is available, testers could begin to ponder whether the responsibility identification should be changed. The modifications of SRS will result in the re-identification of responsibilities. New responsibilities may be added into the original responsibilities when required. The logic change inside a function could lead to breakdown of the original responsibility. The logic change among functions could induce the aggregation of responsibilities. In general, the software maintenance testing methodology is the same to the original development testing.

Today, there are various models of software development. What we discussed in the previous sections is in terms of the waterfall model. For other models, if the resources are available, the testing approach in this thesis can still be applied. The process to identify responsibilities is not a rigid procedure and does not have only a single solution. This gives testers certain flexibility in applying GUI testing.

2.5 Efforts Required for the Proposed Approach

Any testing effort comes with costs. Statistical data gathered over the past years show that testing, analysis, and debugging costs usually consume over 50% of the costs associated with the development of large software systems (Myer [1979]). Therefore, the effectiveness and efficiency are two significant factors in accessing a testing strategy. Usually, the evaluation work needs a baseline for comparison. The GUI testing method proposed in this thesis has two other alternative methods that are possible to be used as the baseline for the evaluation: one is the Complete Interaction Sequences (CISs) method (White [2000]); the other is Planning based GUI testing (Memon [2000b]). The details of both methods have been discussed in Section 1.2.3.

A testing strategy is effective if the tests included by this strategy are likely to reveal bugs in the software. However, without sufficient data and information collected from the real world, it is hard to conclude which strategy is more effective. According to Perry [2000], the information necessary to evaluate the effectiveness of a testing methodology includes:

- Number of tests conducted
- Resources expended in testing
- Test tools used
- Defects uncovered
- Size of software tested
- Days to correct defects
- Defects not corrected
- Defects uncovered during operation that were not uncovered during testing
- Names of defects uncovered
- Developmental phase in which defects were uncovered

Clearly, this involves a significant body of work. Without an understanding of the technical details and the help of special tools, the work cannot be done successfully. Therefore, two empirical investigations are designed to describe the capability of the approach in this thesis. The details of the investigations will be discussed in Chapter 6.

Efficiency is the work done divided by the effort required to produce the work. Efficiency comparisons of different testing strategies also require application to the same program to see how many bugs they find and how much efforts they use to find out these bugs. This is a complex process as well. However, even if it is hard to evaluate the performance of each bug hunting approach, the comparison of all the effort involved in the application of these three testing methods can provide useful

information for the future work. The comparison is based on all the information that we can collect from the relative literature, which is not very precise but gives a rough figure of effort associated these three GUI testing strategies.

Ahead of the evaluation, we will discuss the activities involved in a testing process. This is helpful to understand what kinds of costs are incurred during the entire testing process. Hetzel [1998] describes a model process for performing software testing at any level from unit testing through acceptance testing. The whole process is broken into three major phases: Planning, Acquisition, and Measurement. Figure 2.8 shows the process. Table 2.3. shows more detail activities within each phase.

In Planning, information about the software under test and the project is used to develop testing objectives and an overall testing approach. The output of the phase is a test plan that serves to guide the remainder of the testing activity and coordinate the test levels. In Acquisition, more information about the software (requirements and design), along with previous testing product documentation and data, is used to specify and develop a test configuration for each level of testing performed. The output of this phase is the test set and its documentation. Finally, in Measurement, the test set is executed. The input to this phase is the software to be tested, and the output is test reports documenting the execution and measurement activity along with records of any failures or incidents observed.

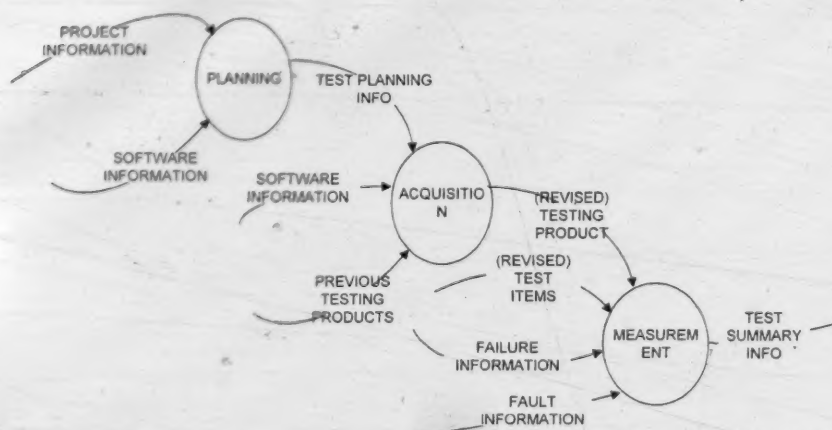


Figure 2.8 Major Phases of Software Testing

Table 2.3 Major Testing Activities

Major Testing Activities:
PLANNING:
1. Plan the general approach
2. Determine testing objectives
3. Refine the general plan
ACQUISITION:
4. Design the tests
5. Implement the tests
MEASUREMENT:
6. Execute the tests
7. Check termination
8. Evaluate results

The size of the test effort should include all test activities undertaken to successfully execute the test. However, the subject discussed here is restricted to testing the GUI of a system. So, in our comparison, effort evaluation focuses on the activities of “designing the tests”, “implementing the tests”, and “executing the tests”.

The three outlined GUI testing strategies have different implementation approaches. The one in this thesis and the one in White [2000] are both conducted manually. The testing process to apply the method proposed in Memon [2000b] is partly automated with the test case generation and oracle generation phases.

On the stage of “designing the tests”, the testing efforts of the method in this thesis are similar with that of White [2000]. Both of them need to gather all related information about the system to divide its GUI into subsystems. In White [2000], a

finite-state machine model is constructed for each subsystem. And in order to reduce the complexity of the finite-state model, strongly connected components and structurally symmetrical components are identified and replaced by super states. It is not clear how much effort is needed to identify the two types of reduction components and whether this reduction may hurt the testing capability of the method. For the current method in this thesis, a flow graph is constructed for each subsystem. The effort focuses on analyzing states of all involved GUI objects, the input and output relationships, and the complex logical relationships of each component. Furthermore, all the subsystems are organized to construct a finite-state machine model on the system tier. The mechanism of the approach in Memon [2000b] is completely different from the other two methods and is based on planning – a well developed technique in artificial intelligence. Given a set of operators, an initial state and a goal state, a planning system produces a sequence of operators that transforms the initial state to the goal state. The sequences of operators are finally translated to test cases for GUIs. The design work includes task identification and operator identification. A task is a test scenario consisting of an initial and a goal state. An operator is a GUI action described by its preconditions and effects. So, it also involves the effort to exploit available information and analyze the functions in a GUI and states of the GUI objects in it.

Without quantifiable effort information, it is very difficult to say which method cost more or less. And it should be noted that for a project, the process to analyze all sources of documents to identify the valuable information for GUI testing is not a completely independent task. Such a process is a part of the work to analyze and design tests for the entire system. Many accomplishments from the work can contribute to multiple testing tasks. The extra effort specific for GUI testing is limited.

The stage of “implementing the test” is the process to generate all test cases for a GUI. This process is automated in Memon’s approach. For White’s method and the one in current paper, test cases are produced according to the constructed models manually. However, efficient algorithms exist for converting the finite state machine model into corresponding regular expressions, which can be transferred to test scripts that can be used by many commercially test tools (Belli [2001]). Technically, it is feasible to

automate the test case generation for these two methods as well. This is also one of our future research directions.

For the execution of the tests, all the three methods can be automated. Memon uses a test executor to run all generated test cases on a GUI automatically. We depend on commercially test record and play tools to automate this process. As discussed previously, if test cases can be expressed in test scripts directly, the execution of test will be very convenient and efficient.

3. Component Tier Testing

3.1. Models of GUI Objects

Developers usually use a *Toolkit* and must follow the pattern of construction laid down by the toolkit vendor. The Toolkits are actually collections of predefined GUI objects that can be called controls, constructs, or widgets. Some GUI objects, such as buttons and labels, are quite simple; others can be very complex, involving data input and logical dependencies. For example, a menu object is always used to construct the hierarchy of a software system. Typically, during development, developers use instances of these precompiled elements stored in a library to build their GUIs.

Fortunately, common understanding prevails as to how these GUI objects should behave in many programming languages. Their operational properties have been standardized over time, making it possible to model these objects for testing. Even if there are some different properties related to similar GUI objects from one Toolkit to another, we can always account for the difference in their models. Apfelbaum [1998] provides example models of 13 primary GUI objects used in conventional GUIs. While this work presents an excellent starting point, we believe that individual testing groups should build their own bespoke models of the fundamental GUI objects found in their own domain. Given this finite set of models as bricks, testers can create a model of a component under test by combining them together. In the next section, we will introduce some examples of models of GUI objects.

3.1.1. Examples of GUI Object Model

Apfelbaum [1998] categories GUI objects into three basic categories: Windows, Menus, and Controls. Windows are composed of many GUI objects such as menus and controls. In our testing strategy, we have different methods to build models for them. The menu is a special GUI object that is used to organize all functions within a GUI. The testing of a menu is actually implemented on both component and system tiers. We focus on the models of some commonly used controls.

A state machine is a system whose output is determined by both the current and past input. It is an engineering application of mathematical models known as finite

automata. The details of finite automata used for our testing of the system tier will be discussed in Chapter 4. A method of using it to model the behavior of a single GUI object is discussed in Apfelbaum [1998]. Here we show only some examples of primary GUI objects including push buttons, radio buttons, check boxes, lists and spinner bars.

- Push button:

A button can display both text and an image. It can be actuated by a mouse or optionally through a keyboard. Generally, developers implement an action listener, which is notified every time a user clicks the button; and the corresponding event handling is stimulated..

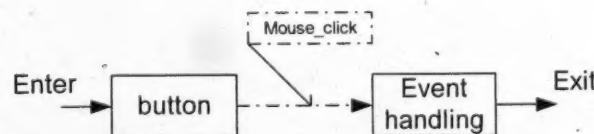


Figure 3.1 The Model of a Push Button

More complex models of buttons involve - what their keyboard mnemonic are, whether they are enabled, selected, or pressed, and so on. The following example is the model of a button with enabled and disabled states.

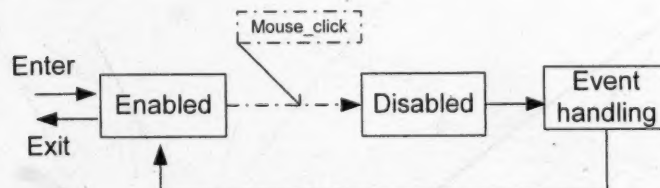


Figure 3.2 The Model of a Button with Enabled/Disabled States

- Radio buttons:

Radio buttons are groups of buttons in which, by convention, only one button at a time can be selected. The destination state is decided by which radio button is selected by a user.

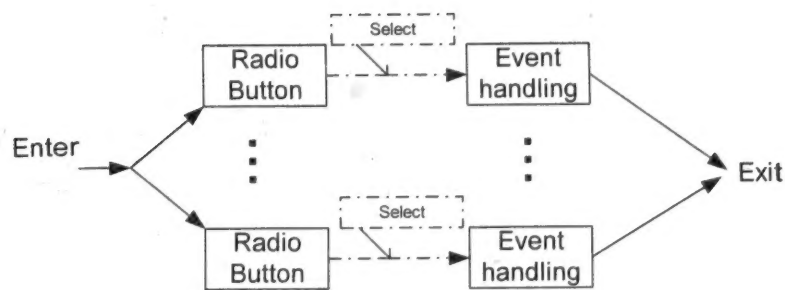


Figure 3.3 The Model of a Radio Button

- Check box:

A check box can be one or more buttons arranged in a row or column, any of which may be in either of two states (checked or unchecked). Once checked, the box remains checked until unchecked. Check boxes are similar to radio buttons but their selection model is different, by convention. Any number of check boxes in a group — none, some, or all — can be selected. A group of radio buttons, on the other hand, can have only one button selected.

The following is the model of one check box. For a group of check boxes, the final state is decided by the combination of the states of all check boxes. The destination state represents the final setting of the Check Box passed to the system under test. Check Boxes are modeled as a chain using the same basic construct.

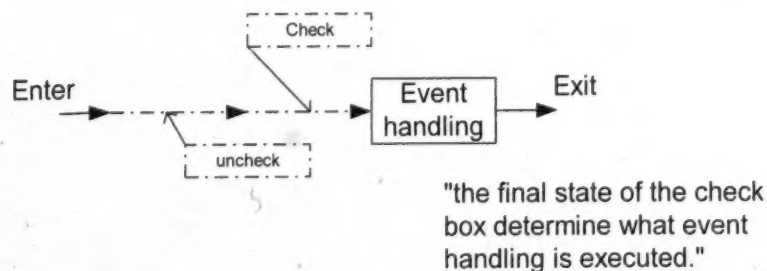


Figure 3.4 The Model of a Check Box

- List

A list presents users with a group of items, displayed in one or more columns, to choose from. Users can select one or more items at once. No matter which selection mode a list uses, it fires list selection events whenever the selection changes.

Typically, a list maintains a string property containing the list of strings displayed in the list box and a value property containing the index into the list of strings that corresponds to the selected item.

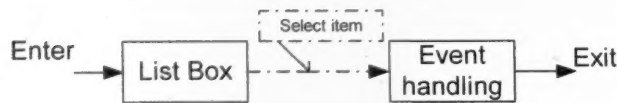


Figure 3.5 The Model of a List

- Combo box

A combo box lets users choose one of several choices. It has two very different forms. The default form is the un-editable combo box, which features a button and a drop-down list of values. Its model is similar to the model of a list. The second form, called the editable combo box, features a text field with a small button abutting it. Users can type values in the text field or click the button to display a drop-down list.

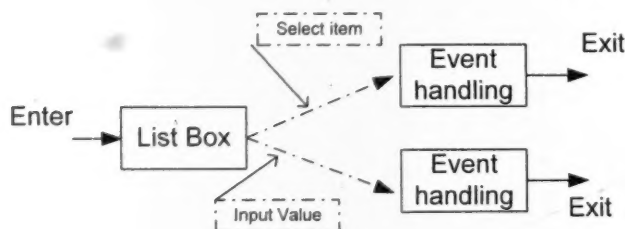


Figure 3.6 The Model of an Editable Combobox

- Scroll Bar

A Scroll Bar is a sliding bar arranged horizontally or vertically to increment a value, or reposition text and graphics in client windows for viewing. A Scroll Bar allows selection based on a controlling policy. Numeric Scroll Bars typically control the selection of real (decimal-type) values. Graphic or Text Window Scroll Bars control panning within a window. A Scroll Bar can be in one and only one state. The state of a Scroll Bar is persistent during the time when a window is being displayed, and may be longer. A selected value remains selected until it is deselected. Scroll Bars transmit data to the System Under Test.

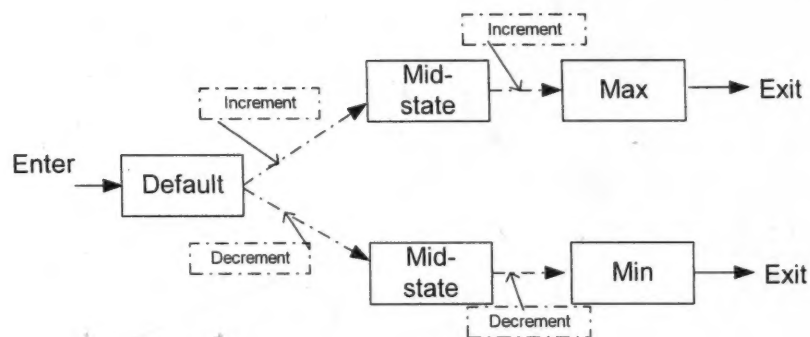


Figure 3.7 The Model of a Scroll Bar

3.1.2. Input Analysis and the GUI Object Model

Actually, the models described above are the basic GUI object models. In many practical situations, testers have to add some special properties to the GUI objects. This gives the tester certain flexibility when applying GUI testing. For example, a list box can have many items in it, and testing each of them is costly. So, testers have to analyze whether all the items are the same so that testing any of them is sufficient or more test cases have to be designed for special items. In practice, the determination depends on the testers' analysis of the GUI object, available testing time and sources for the GUI under test. Each added test case will cause one more path in the model.

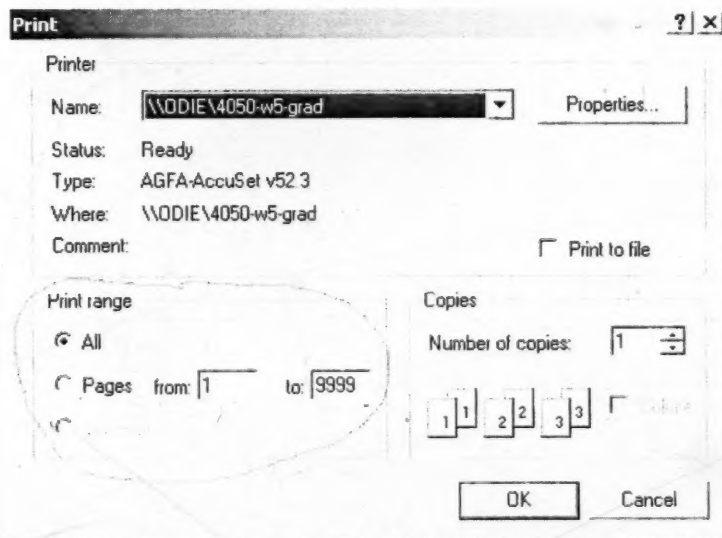


Figure 3.8 The Print Dialog of Notepad Application

The other more crucial point is to combine the result of input domain analysis with the model. We have discussed its importance in Chapter 2 of this thesis. Editable GUI objects allow users to input values to the underlying applications. Hence, test data from different divisions are necessary to verify whether the GUI works well under varied situations. Furthermore, some illegal and erroneous inputs have to be applied to the GUI to trigger all warning or error messages; Figure 3.8 shows the print dialog of the Notepad application to illustrate this concept. This dialog includes a group of radio buttons to set the print range. The user can select to print out all pages of a file or a certain page range. The *from* and *to* text boxes beside the *Pages* button make this radio button different from the basic model. The text boxes allow the user to input the page numbers of the print range. From the syntax point of view, the valid range for these two numbers is 1-9999, and the value of *to* should be greater than or equal to the value of *from*. When users click the OK button, the program checks both the values according to the rules before creating the printing job. By using the equivalence partitioning technology in Myers [1979], we identify one valid equivalence class ($1 \leq \text{input} \leq 9999$) and two invalid equivalence classes ($\text{input} < 1$, $\text{input} > 9999$) for each of the text boxes. For the logic between the value of *to* and *from*, we identify three conditions: $\text{to} > \text{from}$; $\text{to} = \text{from}$; and $\text{to} < \text{from}$. All of these results will be added to the model of this radio button. So the Pages radio button branch is composed of multiple branches as presented in Figure 3.9. Different branches may invoke different message boxes. According to the model, we will develop the test cases shown in Table 3.1 to verify the responsibility of the two text boxes. This model only illustrates the process of data verification. In practice, whether the print job can be created and finally processed involves another factor that whether the page values fall in the actual range of the file. However, we do not consider this factor in the model because it does not affect the GUI.

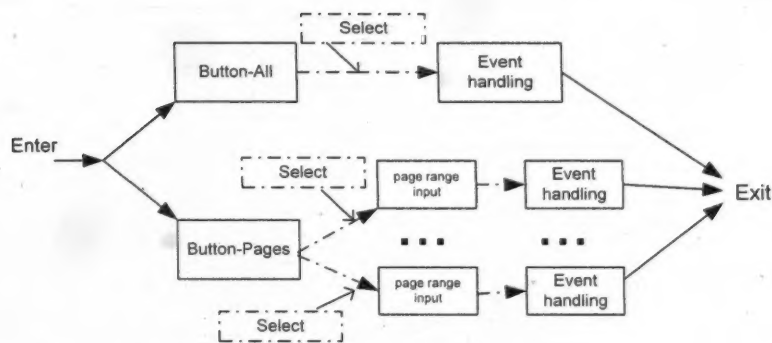


Figure 3.9 The Model of Print Range Radio Buttons

Table 3.1 Test Cases for Print Range

Input		Result
From	To	
1	9999	Valid
0	2	Invalid
2	1	Invalid
1	0	Invalid
1	10000	Invalid
10000	-	Invalid
-	1	Invalid
1	-	Valid
1	1	Valid

3.2 Flow Graph on the Component Tier

As described in Section 2.1, dividing an entire GUI system into simpler independent subsystems is necessary and reasonable for GUI testing. In our testing strategy, a subsystem is called a component, which is regarded as a unit of testing. Before defining all components of a GUI, testers have to identify all responsibilities defined

in the GUI software. A responsibility is composed of a sequence of activities conducted by users on related GUI objects to complete a task in an environment. To accomplish this task, testers have to explore and analyze all the available information sources that they can gather concerning the system. The possible sources include the requirement specifications, GUI design documents, user manuals, and even the code of the system. If necessary, testers may interact with GUI systems directly. The process to identify responsibilities has been discussed in depth in Section 2.4.

For each identified responsibility, a corresponding component is defined. First, it is necessary to explore and analyze all GUI objects within the system, such as windows, dialogs, icons, menus, etc. For each of these objects, testers have to identify all possible selections available, all possible states, possible links to other GUI objects, and possible links to other non-GUI components of the system. The responsible GUI objects are then organized to form a component of the GUI. Hence, a component is a collection of GUI objects, as long as they contribute to the implementation of the responsibility. One GUI object may work for multiple responsibilities; so it may be present in multiple corresponding components.

The testing on the component tier aims at verifying whether the behavior of an individual component meets the requirements for the corresponding responsibility. Some responsibilities are very simple and straightforward. As soon as users invoke the corresponding GUI objects, their event handling programs start to run and complete the task without any more inputs or parameters from the users. For example, the *Redo* and *Undo* responsibility in the Notepad Application. Checking whether they work well independently is straightforward. For these components, testing efforts focus on checking whether they work properly in different circumstances. Doing so involves testing the interactions among the components. We leave this testing to the system tier testing, which is based on a finite state machine model.

On the other hand, many responsibilities are complex processes involving complicated logical relationships and multiple steps; for example, the "Open an exiting text file" responsibility in Notepad includes several steps:

- Checking whether the current text area needs be saved.

- Getting the drive and file name
- Loading the contents of the file in the text area.

Usually, the components associated with these responsibilities include various GUI objects. Different inputs to the objects result in different execution paths through a component. The input space of GUI objects and the logical relationships among them are explored and analyzed. Our testing objective on this tier is to verify whether the component works properly through different paths.

Hence, all components that need inputs from users, such as parameter sets and decision-making steps, are tested on this tier. Generally, the execution of these components starts from GUI objects in the top-level frame, such as the application's main window; these GUI objects invoke sub-frames comprising more GUI objects, which in turn can invoke still more frames. These objects can be visualized as a tree structure, each one having a single parent and a number of children. When users interact with a GUI, which path they go through depends on not only the event environmental conditions but also what kind of inputs the users give to the involved GUI objects. Unfortunately, this tree structure provides insufficient information to carry out scalable testing. A test-ready model should contain enough information to produce test cases for its implementation automatically. In Section 3.1, we discussed how to model a single GUI object and how to combine the input space analysis result with its model. Now we need to organize all related GUI object models together to represent the components. In our testing strategy, a flow graph is created for each component. It is a diagrammatic representation of the possible alternative control flow paths through a component.

In traditional branch testing, it is convenient to examine a directed graph representation of a program, which shows the flow of control through the program by using a network of nodes and edges. In this thesis, a flow graph is similar to the directed graph, but it is used to show the control transfer from one GUI object to another. It is a combination of all GUI objects inside a component and the logical relationships among them. A node in a flow graph represents a GUI object or a decision-making step according to the environmental conditions at the moment instead of a block of code. Edges on the flow graph represent users' GUI activities on

GUI objects that cause the invocation of an object and/or the closing of another. Each flow graph has one start node — the state of the component before the execution, and one end node — the state of the component after the execution. Each path, which is composed of the edges from the start node to the end node, is actually a sequence of events causing the component to transit from an original state to a new state. The path can provide all the information necessary for the test case generation.

3.3 An Example of a Flow Graph

In this section, we present an example to show how to create a flow graph in detail. The GUI of Notepad, which has been introduced in section 2.3, is used as the example. We analyze all responsibilities implemented in it and then defined the following eight components:

- Open an existing text file.
- Create a new text file.
- Edit a text file.
 - Redo a canceled action
 - Undo the last action
 - Cut
 - Copy
 - Paste
 - Wrap line
- Find a word in the text.
- Set the attributes of the font.
- Save a text file.
- Print out a text file.

- Exit the application.

The purpose of "Create a new text file" component is to create a new file with a blank text area and a currently unspecified file name (upon starting Notepad, it enters into this state). We will show a detailed model of this component. Before the construction of its flow graph, the involved GUI objects necessary to create a file are the following:

- Menu item File-new (shown in Figure 2.2) can invoke the "Create a file" function.
- The Confirm save message box (shown in Figure 3.10) allows users to select whether to save the unsaved contents in the text area or not.
- The Save dialog (shown in Figure 3.11) collects all the information necessary to save the contents in the text area to a file.
- The text area (shown in Figure 2.2) is the area that shows the contents of a file and the users' inputs. When completing the "Create a new text file" component, the text area is set as blank with an unspecified title.

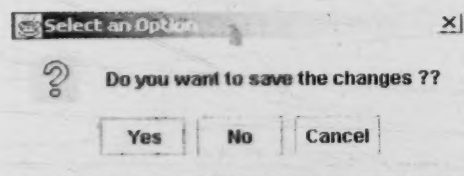


Figure 3.10 The Confirm Save Message Box

All these objects should appear in the flow graph. For each of these objects, all possible selections available, all possible inputs, all possible links to other GUI objects, and all possible links to other non-GUI components of the system should be identified. As discussed in Section 3.1, state models for these GUI objects are created by combining all the associated information. Some of the objects are sub-frames composed of more GUI objects. For a simple sub-frame, such as the Confirm Save Message Box, which has only three buttons without any complex interaction, it is convenient to integrate their models into the flow graph directly. For complex sub-frames, such as the "Save a File Dialog" (shown in Figure 3.11) with many GUI

objects and complex interactions among them, we are required to create their own sub-graphs.

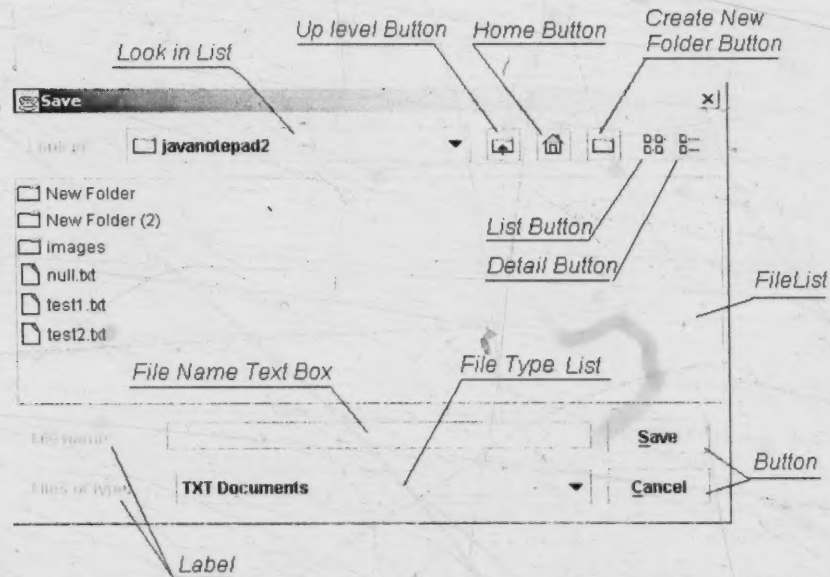


Figure 3.11. The Save a File Dialog

Furthermore, we explore the preconditions and the possible outputs for each component because how users interact with the component depends on not only their subsequent GUI activities on the objects but also the previous activities; all the pre-activities actually bring the GUI to a state that we call the original state of the component. After users apply activities on the GUI objects to invoke the underlying application to run certain tasks, the GUI is brought to a new state. The state of a GUI can be defined as the collection of all its attributes. Though the number of these attributes is usually huge, for the application of a responsibility, only a few of the attributes are involved. Actually, the values of these attributes can affect the execution path of the component. These values compose the preconditions of the flow graph. On the other hand, for the new state of the GUI, after the sequence of users' activities, we are only concerned with the attributes that can be changed during the execution of the component. All these attributes actually represent the outputs of the component.

The precondition attributes and the output attributes for "Create a new file" are summarized in the Table 3.2

Table 3.2 Summary of the Attributes for "Create a new file"

Precondition Attributes	Output Attributes
• The text area is empty or not • The text area has no unsaved contents or not. • The original file has file name or not	• The text area is blank and the title is "untitled" file • Save the original content of the text area to a file.

With all necessary information at hand, the flow graph for "Create a new text file" is created as shown in Figure 3.12. In the flow graph, a node can be represented as a rectangle or a diamond: a rectangle represents a model of a GUI object; a diamond represents a decision-making step according to the related attributes of the environments. It should be noted that some nodes of the flow graph are complex sub-frames, such as dialogs windows. For these dialogs or windows, sub-graphs will be further created by using models of associated GUI objects. In many situations, testing different event-sequences inside a sub-frame is not very meaningful. For example, a dialog, regardless of its type, is always composed of many GUI objects designed to solicit information from users. These objects are located on the same tier. The possible sequences of activities performed on them may be huge, but most do not result in significantly different outputs. Before a user invokes a confirming action, which is generally to click on an "Ok" button, the software does not pass the selections or data to the underlying code. So no matter by what sequence the user applies activities on the GUI objects, the underlying code run according to the final states of the GUI objects before the confirming action. In this case, the permutation of varied states of the GUI object models is much more important for testing than the permutation of event sequences on them. Section 2.2 provides an example to describe this.

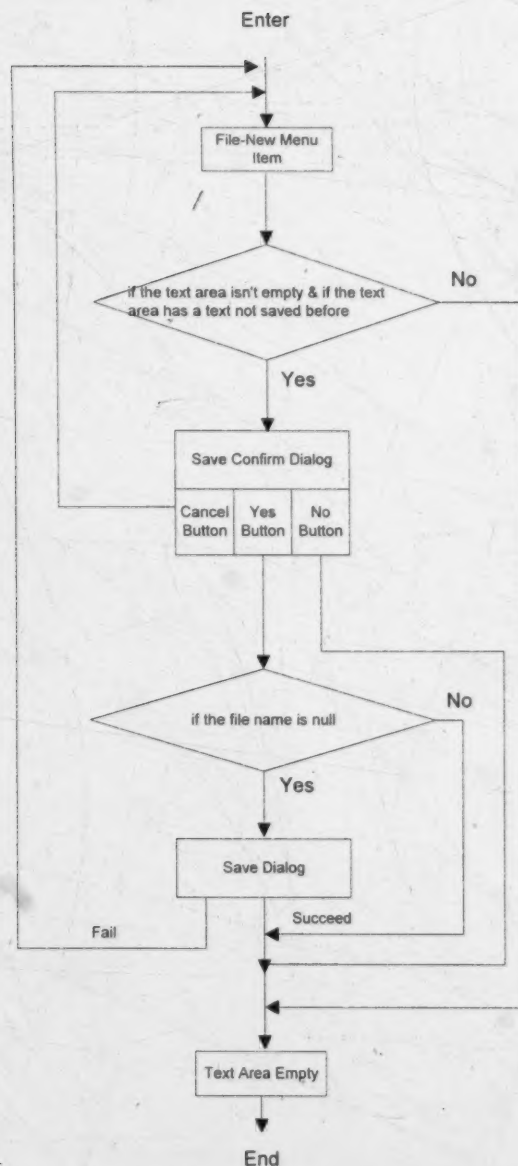


Figure 3.12 New Component Flow Graph

Figure 3.11 shows the “Save a file” dialog within the “Create a text file” component. The dialog is composed of seven buttons, two drop-down lists, one list, one text box, and some labels. When creating the testing model, we consider only the functional part of the GUIs. Their aesthetic conditions, such as GUI objects’ size, positions and types, are not involved in this testing. These attributes of GUIs can be verified by using other static testing techniques, such as walkthroughs, inspections, reviews and

so on. The website <http://members.tripod.com/~bazman/checklist.html> gives a long checklist on this topic, covering many aspects of GUI verification. In the current testing strategy, these static GUI objects, such as labels, are not included in the testing model. Another issue is that the "Save a file" dialog is already implemented by Java in the FileChooser class. There are five buttons in the right top corner. They all work well in Java JDK 1.4. But the "Detail" button and "List" button do not work in Java JDK 1.2. Because the Rational Robot that we used in the empirical experiment for this thesis can only support up to Java 1.2 to record testing scripts, I did not include these two buttons in the experiment. As discussed in Section 3.1, a state machine-based behavioral model can represent each GUI object. Given these models as bricks, testers integrate them and combine the logical relationship among them to create the model of the dialog. Figure 3.13 is a simplified graph of the "Save a file" dialog. The dash box includes all GUI objects in the dialog, except the Cancel and Save buttons, which cause the dialog to terminate. Each rectangle in the diagram represents a model of an object. Each GUI object's model can have one or more paths to the corresponding states. Our test cases will cover all the paths of the objects. This model can replace the node of "Save dialog" in Figure 3.12.

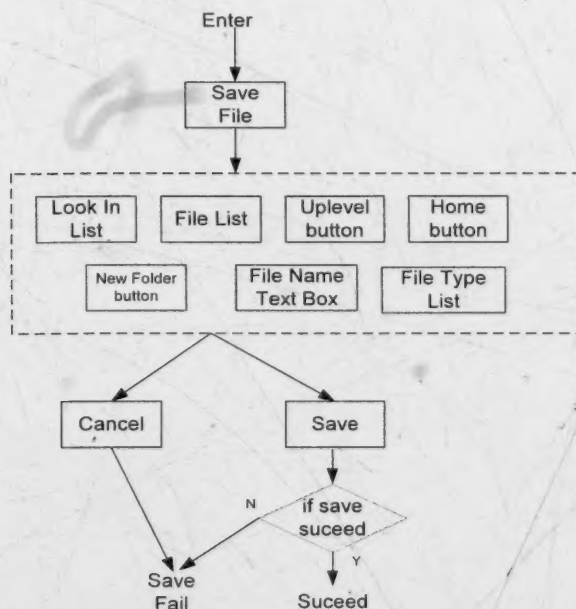


Figure 3.13 The Sub-graph of Save a File Dialog

At this point, the flow graph is created completely. From the start node to the end node, involved edges consist of an execution path of the component. Actually, all the edges are a sequence of events causing the GUI to transit from an original state to a new state. A path can be translated to a test case for the component.

For other components in Notepad, similar processes are followed to construct their flow graphs. Their test cases are generated respectively. Our testing objective on this tier is to verify whether all these functional implementations can meet their requirements under varied inputs. To some extent, testing on this tier belongs to component testing, which is a very important aspect of testing for a project. A very large percentage of errors can be found at this level. So further testing on the system tier is based on a trusted component tier. In the system testing stage, the testing concerns will focus on the interactions among the components.

In practical industrial testing situations, testing often occurs under tight cost and time constraints. The decomposition of a GUI system into responsibility components is more flexible to allow testers to tailor the amount of effort involved in the testing to accommodate their economic requirements. This value-oriented approach to test case management obviously parallels with the use of operational profiling (Musa [1998]). In addition, test cases generated on the component tier can be tailored to be reused on the system tier testing.

4. System Tier Testing

4.1. Introduction

The two-tiered GUI testing strategy proposed in this thesis applies testing on different abstract levels of GUI software to deal with different challenges associated with this problem one step at a time. On the component tier, we focus on analyzing the complicated input domain of each component and verifying component behaviors under varied input combinations. The system tier testing tries to verify that the total GUI system satisfies all of its functional, quality attribute and operational requirements. It primarily demonstrates that all components in the GUI system can work together to fulfill the requirements.

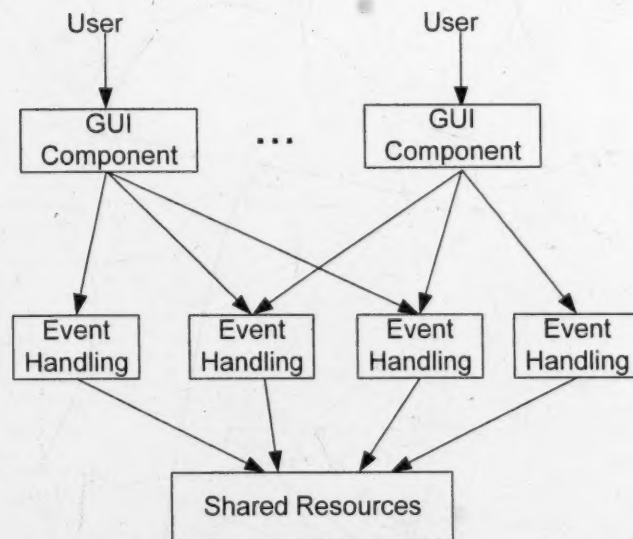


Figure 4.1 Interactions among Components

As discussed in Section 2.2, the inputs to a GUI are actually sequences of events conducted by users on GUI objects. Each event has a corresponding event handling process; different sequences of events mean different sequences of execution of their processes and may cause different outputs from the GUI or lead the GUI into different states. In most situations, the components do not implement their responsibilities in isolation but within a context. Usually, these contexts partially overlap with each other and share resources, causing the complex interactions among the components.

Figure 4.1 describes the relationships. On the system tier, each tested component can be treated as a trusted unit. A sequence of these units represents a path by which users go through the whole system. Whether a GUI works properly under these sequences or not is our main testing concern on the system tier. A finite state machine is used to model the behavior of the entire GUI. Section 4.2 will discuss the finite state machine in detail. In Section 4.3, we will further describe how to model a GUI by using a finite state machine.

4.2. Finite State Machine

For GUI testing, a typical scenario is that a tester applies an input to a GUI and then appraises the result; the tester then selects another input, and reappraises the next output again. At any given time, a tester has a specific set of inputs to choose from. This set of available inputs varies according to the concurrent state of the GUI software. This characteristic makes state-based models logically suitable to describe GUIs.

A finite state machine, also known as a finite automaton, contains a finite number of states and produces outputs on state transitions after receiving inputs. There are stable and mature theories on FSM and its variations that have been widely used to model the behavior of systems in varied areas such as sequential circuits, lexical analysis, and real time process control. Using finite state models in the design and testing of computer hardware components has been long established and is considered a standard practice. For software testing, system behavior is modeled as a finite state machine in which the event sequences trigger a transition from one state to another. A path in a FSM represents a test case. Several previous research efforts have recognized that the use of finite state machines can facilitate at least part of the user interface testing.

Chow [1978] addressed the use of finite state models to design and test software systems. He proposed modeling the control structure of a software system at the design level by using a finite state machine. This method was applied to software that can be described by stimuli and operations: stimuli were inputs from the world outside the software system; operations were events caused by the software system.

The generated tests can check the correctness of the control structure at the design level of many software systems.

Recent practices include those of Beizer [1995], who introduced the finite state model as one of the essential models for black-box testing and a process to create these models. He concluded that the FSM could be applied to many types of software including menu-drive software, object-oriented software, protocols, device drivers, installation software, etc. Binder [2000] analyzed how to develop testable state models and state-based test suites for object-oriented software systems. Bochmann [1994] applied FSM to communication protocol testing. In Apfelbaum [1998], a state-based model was created to describe the behavior of a single GUI object. We extend their model by adding more information from the results of logical analysis, such as equivalence class analysis, boundary value analysis, and special values analysis. As discussed in Section 1.2.3, White [2000] described another important FSM application for software testing, especially for GUI testing. In a GUI, Complete Interact Sequences were defined and modeled by finite state machines. Test suites were generated according to these models.

In general, we are able to specify a system's behavior in terms of states when the subjects are small-sized system. For example, when the FSM of a GUI is created, each user action leading to a new state or triggering a transition is considered. However, for a large and complex software system, the number of distinct states and transitions is extremely large resulting in a major limitation in applying FSM for software testing. In order to solve this problem, variations such as variable finite state machine (VFSM) models, have been proposed by Shehady, etc. (Shehady [1997]). A VFSM is similar to a normal FSM but is augmented to allow for a number of global variables, each of which can assume a finite number of values during the execution of an input sequence. The value of each variable is used to determine the next state and the output in response to an input and can be modified by a transition. In many situations, a VFSM has far fewer states than the corresponding normal FSM and maintains its expressive capability. For the approach in this thesis, we avoid the scalability problem in a different way. In the rest of this section, we will introduce the foundation of the FSM model. How to use the FSM in the system tier testing is discussed in the next section.

Finite state machines are ideal models for describing sequences of inputs to systems. This fact, combined with a wealth of graph traversal algorithms, makes generating tests less of a burden than manual testing (El-Farand [2001]). On our system tier, a finite state model is created to describe the interactive behavior of the components in a GUI, and corresponding test data are generated to achieve the testing objective on this level. The literature on finite state machine theory is rich in terms of designing, manipulating, and analyzing finite state machines. The following discussion is based on information from Gill [1962], Hopcroft & Ullman [1979], Beizer [1995] and El-Farand [2001].

According to Gill [1962], the exact definition of a finite state machine is:

A finite state machine M is a system with:

a finite input alphabet $X = \{\xi_1, \xi_2, \dots, \xi_p\}$,

a finite output alphabet $Z = \{\zeta_1, \zeta_2, \dots, \zeta_q\}$,

a finite state set $S = \{\sigma_1, \sigma_2, \dots, \sigma_n\}$,

and a pair of characterizing functions f_z and f_s , given by:

$$z_v = f_z(x_v, s_v)$$

$$s_v = f_s(x_v, s_v)$$

where x_v , z_v and s_v are, respectively, the input symbols, output symbols, and the state of M at time t_v ($v = 1, 2, \dots$). In short, a FSM can be represented by a set of inputs, a set of outputs, a set of states, and two functions: one is the next state function that maps pairs of inputs and states to next states; the other is the output function that maps pairs of inputs and states to outputs. A finite state machine can be in only one state at any one time. The occurrence of a transition from one state to another is exclusively dependent on an input in a machine.

Usually, a FSM is a *deterministic* finite state machine; its characterizing functions are not subject to any uncertainty. In a non-deterministic machine, there can be more than

one transition from a given state for a given possible input. Generally, a non-deterministic machine is implemented by converting it into a deterministic machine.

Currently, there are many ways to represent a FSM. A state transition diagram is one of the most popular ways; it is a directed graph whose vertices correspond to the states of the machine and whose edges correspond to the state transitions; each edge is labeled with the input and output associated with the transition (Lee [1996]). Figure 4.2 is a simple example of a state diagram with four states, and two types of inputs. The S0 state is the starting state and final state as well.

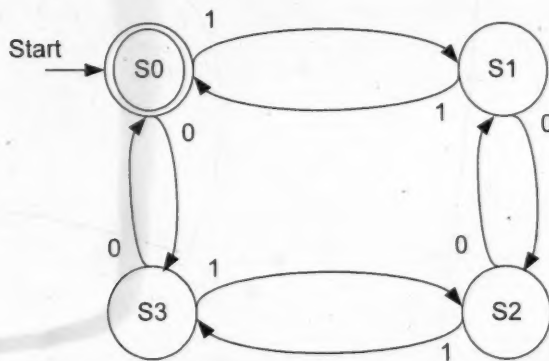


Figure 4.2 An Example of a State Diagram

Actually, there are many variations upon this definition that are produced for different objectives. For example, the model may be restricted to having only one starting state and one final state; others may have multiple starting and final states. In some state machines, a transition is dependent, in addition to the current state and the application of an input, on some conditions external to the machine. Another very useful extension is that of a hierarchical machine, the states and transitions can be extended to represent other finite state machines. Hierarchical models allow complex behavior to be decomposed into simpler lower level models (El-Farand [2001]).

There are two useful variations of finite state machine models: the Mealy model and the Moore model. In a Mealy model, outputs are associated with transitions. Whenever a certain transition between two states is made, a certain output is generated. Figure 4.3 shows an example of a Mealy model. In the alternative model, the Moore model, the outputs are associated with states. Whenever a certain state is

entered, a certain output is generated. Figure 4.4 describes the corresponding Moore model for the same machine shown in Figure 4.3.

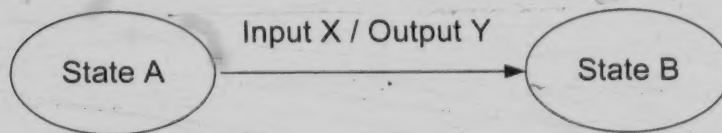


Figure 4.3 A Mealy Model Example

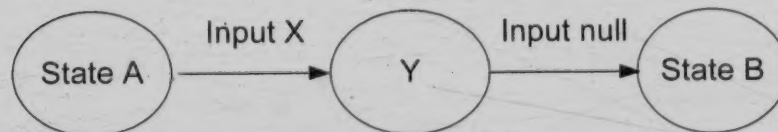


Figure 4.4 A Moore Model Example

For software testing, the Mealy machine is more popular. The reasons for this are that, first, comparably, Mealy models have fewer states than Moore ones and are therefore simpler. The number of states in a model may affect the complexity of the tests significantly. Second, the states in a Mealy model are stable. The Moore model always has transient states, which may make testing more difficult. Finally, the Mealy model is closer to implementation. Programmers who do explicit finite state implementations usually base them on Mealy rather than Moore models. If the test and design models differ, this is a source of test design bugs.

The selection of the states of a FSM in any given problem is neither unique nor necessarily simple. And there are no available formal rules for this selection. Hence, this choice really depends on the practitioners' domain knowledge and acquaintance with the system at hand to build a propiate model.

However, for our system tier testing, the testing objective is not to identify well-defined states for a GUI but to evaluate the operating sequences of the components that are likely to occur in practice. Certain variations have to be applied to the basic model to make the FSM adaptable to our testing. The model is required to integrate all the components identified during the previous testing stage into a system. And this model can represent the way by which users interact with GUIs. By inputting event sequences, users can move from one component to another.

4.3 The Model for the System Tier Testing

The testing strategy on the system tier is to find the most effective and efficient input event sequences, which are applied to a GUI to check whether it produces correct responses to users and whether all its components identified during the previous testing stage work together. Although finite state machines represent a very powerful way of describing and implementing the logics for applications, when it is applied to GUIs some variations to the basic model are necessary because of the special characteristics of the GUI software.

As discussed in previous sections, the number of distinct event sequence inputs to a GUI may be huge and result in a large number of GUI states. It makes the building, maintaining, reviewing, checking and testing of the FSM models very difficult or even impossible. For large and complex systems, state explosion creates a serious challenge to apply model-based testing. This challenge is not unique to the finite state models. And for GUI software, this problem is more prominent. Basically, there are two ways to deal with state explosion: abstraction and exclusion (El-Farand [2001]).

Abstraction involves the incorporation of complex data into a simple structure. For example, a dialog box requiring the entry of multiple fields of data before clicking the OK button can be modeled as two abstract inputs: *valid data entry* and *invalid data entry*. All the fields are treated as a structure. If the entire set of the data is valid, the transition to the correct state is made. When the data are invalid, the model represents enough knowledge to transition to the error state. The downside of using abstraction is that it always has the effect of losing some information. In this case, any possible error associated with the entry is simplified as invalid data entry. We lose the information about exactly which field is misused. But when used wisely, abstraction can retain important information and ignore superfluous or irrelevant information without loss of effectiveness. The other way to deal with state explosion is exclusion that means dropping information rather than abstracting it. This is done by decomposing the software behavior into multiple models. Each model will contain certain inputs and exclude others. Hence, each model covers different aspects of a system.

The testing strategy proposed in this thesis has a similar philosophy to exclusion, which applies GUI testing on two tiers: the component tier and the system tier. On the two tiers, different models are created to represent different profiles of a GUI. On the component tier, the low level details of GUI objects and functional components are verified with respect to the flow graph model. Hence, on the system tier, testers deal with the components as trusted components. The finite state model on the system tier focuses on the interactions among the components instead of the details inside them. Doing so significantly simplifies the model on the system tier and avoids the state space explosion without losing the details that are required to reveal bugs. Therefore, we require that the inputs to the system model are not trivial event sequences but the complete event sequences that can achieve the functions of a component.

The other challenge of applying finite state models for GUI testing is state definition. A state in a FSM at any given sampling time is identified with respect to the variables of the system. Usually, a GUI has lots of attributes which are likely to affect its state. To explicitly define the states of a GUI is not an easy job. However, the testing objective for a GUI system is not to check its rigid states and then verify every single transition but to evaluate the external behaviors of the GUI and the operating sequences of its components that are likely to occur in practice. What we need is a model to integrate all the components identified on the previous testing stage into a system, which can represent the way in which users interact with GUIs. By inputting event sequences, users can use the functions provided by the software in the order that they want. Hence, the model on the system tier facilitates the generation of effective event sequences to verify transitions from one component to another.

The model used in the current thesis is a variation from the basic FSM model. We will give an example to describe how to make the FSM adapt for GUI testing. Figure 4.5 shows a typical state transition diagram, in which a set of rigid states (0,1,2), input (a and b) and corresponding output is defined. The edges in the diagram represent the functions that map the states and inputs into the next states.



Figure 4.5 A Typical State Transition Diagram

The model used on the system tier should concentrate on the event sequence inputs to the system to verify the operational behavior of the system. To reach this target, the basic model in Figure 4.5 is simplified to only include input event sequences on the diagram. This method is, also used in White [2000] and Fevzi [2001], has its root in Myhill graphs (Myhill [1957]). The new model is called as Event Sequence Diagram and shown in Figure 4.6. A state on this model is characterized by a set of input events and its responses, which determine the invocation of transitions from one state to the others. The input events here are the operations on GUI objects by input devices, such as a mouse. Actually, in an Event Sequence Diagram, the term event sequence and the state are interchangeable. This simplification solves the state definition problem for GUI software without weakening the computational power of the model (Fevzi [2001]). With respect to the Event Sequence Diagram, the test cases for a GUI are generated in a systematic and scalable manner.



Figure 4.6 The Event Sequence Diagram

Each state in the event sequence diagram can be extended to a sub-graph that represents another finite state model. This hierarchy allows complex behavior to be decomposed into simpler, lower level models. For complex GUIs, this abstract is necessary. Or the number of event sequences would be so large that the analysis of the system will be impossible for all practical purposes. The difference between the model in this thesis with the models in White [2000] and Fevzi [2001] is that we apply this model on the system tier assuming the details about GUI objects has been tested on the component tier, as described in Chapter 3. The model on the system tier testing is created according to the components identified in the previous stage. The input event sequences have to complete certain component functionality instead of trivial operations on the GUI objects of a component. The models in White [2000] and Fevzi [2001] try to cover all low-level information such as user actions on GUI objects and actually have to exclude some details that are likely to reveal bugs in GUIs. The behavior of a single GUI object is over-simplified. Each GUI object is represented as a node on the state diagram of a GUI. In practice, each GUI object can

have many states and under different inputs, its output and next state can change. Overlooking the characteristics of these GUI objects can cause bugs to be missed.

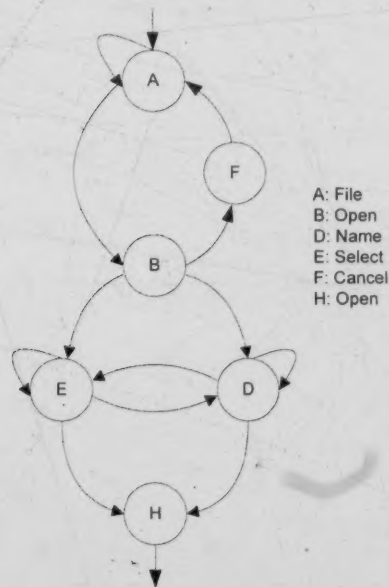


Figure 4.7 Event Sequence Diagram of "Open File"

Figure 4.7 shows the finite state model of "Open File" that is a part of MS WordPad application demonstrated in White [2000]. In the model, the behavior of all GUI objects is simplified to a single situation. For example, the user action of "Name" in the model means that users input a valid file name in the open dialog. But in practice, users may input valid or invalid file names and the system may present different behavior under these different inputs. Only considering the valid situation makes the testing model incomplete and inaccurate reflection of the implementation to be tested. In Fevzi [2001], event sequence diagrams are extended by adding illegal interaction pairs. However, the invalid situation remains missing in their model. Furthermore, most of the illegal interaction pairs are infeasible to apply to GUIs and they actually do not represent typical user faults in the interaction with GUIs. For example, the fault event sequence diagram of "open file" is created by adding edges in opposite, adding iterations to vertices wherever none exists, and adding two-way edges between vertices wherever no edges exist. The fault event sequence diagram is shown in Figure 4.8

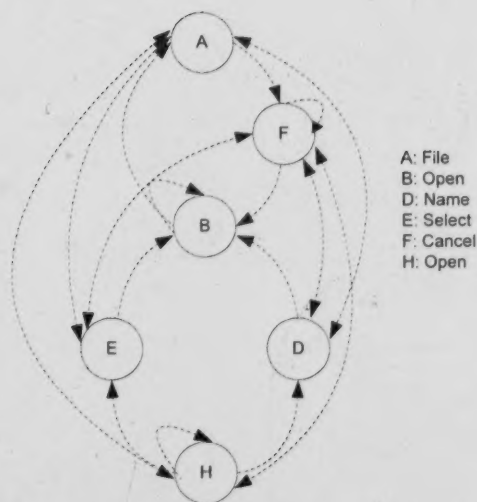


Figure 4.8 Fault Event Sequence Diagram of "Open File"

In the diagram, A represents the File menu and B represents the Open menu item. E, D, F, H are GUI objects on the open dialog that is a model dialog. So all edges between A and E, D, H, F and all edges between B and E, D, H, F are infeasible to be applied. The same things happen to H and F, which are operations to terminate the open dialog. Hence, the edges from H and F to the GUI objects of the open dialog are impossible to apply too. Among the 22 edges on the diagram only 3 are feasible for GUI testing.

In the approach of this thesis, "Open File" is defined as a responsibility, which is represented by a flow graph shown in Figure 4.9.

This flow graph includes all GUI objects that are involved in the "Open File" component. Each of the GUI object is not treated as a single node. It can be further represented by sub-graph. Different states of it can be taken into consideration during modeling. The details of building the flow graph are discussed in Chapter 3. The event sequence diagram is used at the abstraction level of the entire GUI system and is constructed with all event sequences that can implement the functions of the previously identified components as inputs. Hence, in the event sequence diagram, the event sequences are represented in terms of the components' name. As discussed in Section 3.3, for our example application – Notepad, thirteen components were defined. Each of the components has been thoroughly tested on the component tier and can be

treated as a trusted atom. As discussed previously in Section 2.4, for a complex GUI system, the number of its components may be large, causing the structure of the finite

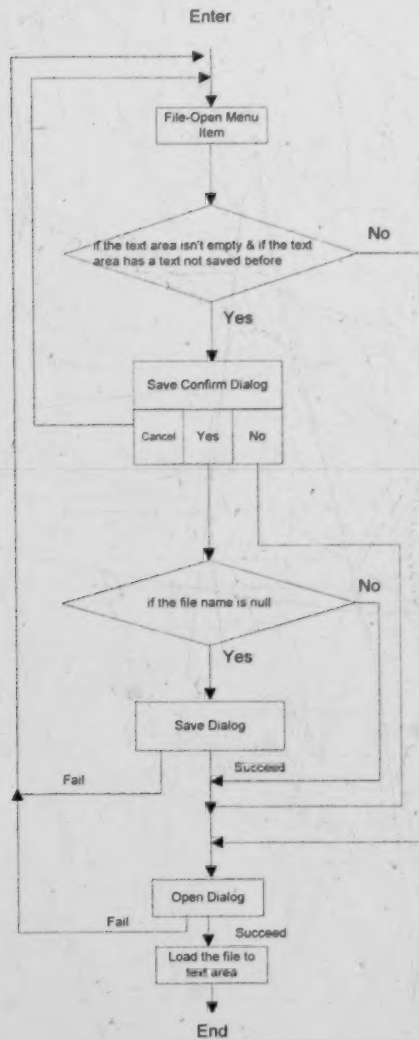


Figure 4.9 The Flow Graph of "Open File"

state model very complicated and the testing cost too high taking the effectiveness into account. A way to deal with this problem is to group certain components together as a compound state of the finite state model, which can be represented as a sub-finite state machine as well. When creating the system event sequence diagram of the Notepad, the thirteen components can be organized into five states:

- Open File

- Edit File
- Save File
- Print File
- Exit

Figure 4.10 shows the event sequence diagram of Notepad. All edges between the states cause the transitions from one state to another in it. Meanwhile, users navigate through all the features provided by the GUI system. To some extent, this model represents the hierarchy frame of a GUI, which organizes all components together. This modeling process is done manually and requires test designers with some practical experience and domain knowledge of the GUI system under testing.

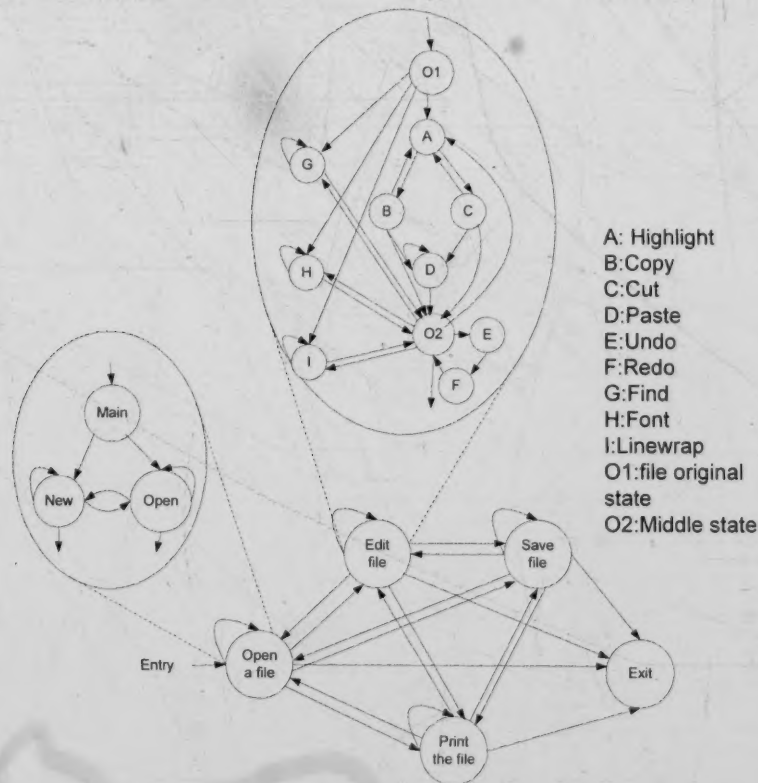


Figure 4.10 The Event Sequence diagram of Notepad

Once the testing model has been constructed on the system tier, we can identify entire sets of event sequences as edges of the diagram. Corresponding test cases are produced to satisfy edge coverage that tester define at this tier. In the next chapter, we will discuss defining coverage criteria for GUI testing.

5. Coverage Criteria

5.1. Introduction

One important question about software testing is when to stop testing. To answer this question, we will introduce another key issue for testing, coverage criteria, which are sets of rules to measure whether a test suite is adequate. In fact, coverage is an excellent feedback mechanism for software engineers. It provides useful guidelines that help engineers tailor their effort. In practice, a great variety of coverage criteria exist. Generally, a certain type of coverage is a measure of the effort to detect a certain class of potential errors. For example, 100% statement coverage does not only mean that every line of code has been executed; it also means that software has been tested for the bugs that can be revealed by simple execution of a line of code. Hence, when researchers talk about the completeness or thoroughness of software testing, the terms are not meaningful without reference to an explicit coverage criteria. However, research on coverage criteria for GUIs has seldom been addressed. The only known work is the event sequence based coverage criteria for GUI testing proposed in Memon [2001]. In the following part of this section, we firstly introduce the coverage criteria defined in Memon [2001] and then describe the criteria used in this thesis.

5.2. Coverage Criteria of GUI Testing

Usually, coverage has a very close relationship with test case generation, because each type of coverage implies an operational definition for a complete test suite. A testing model always suggests a coverage goal to generate corresponding test cases. So all elements of the model that are likely to contain defects are exercised during testing.

To define an appropriate coverage goal for a model, some evaluation criteria should be considered in advance. First, a coverage metric should be achievable. It should be possible and practical to achieve 100% coverage. Any value less than 100% requires investigation to determine why less than 100% has been achieved. Second, software has to be retested many times throughout its lifetime. Test data required to achieve 100% coverage therefore have to be maintainable. Changes required of test data should not be disproportionate in scale to changes made to the code. Finally, an ideal

criterion to evaluate a coverage metric should be effective at detecting faults in software. To measure the effectiveness of each coverage metric requires extensive data collection from software tested using the entire range of coverage metrics. Hence, it is difficult to quantify the effectiveness of each metric. So thoroughness is used as an index of effectiveness. During the process in which test data are designed to fulfill a metric, the thoroughness is assessed. A higher thoroughness score is attributed to metrics that demand more rigorous test data to achieve 100% coverage score (IPL [1997]).

GUI testing differs from the underlying application code in its level of abstraction. Mapping between the appearances or events of a GUI and the underlying code is not simple or straightforward. As discussed in Section 2.2, the input to a GUI consists of a sequence of events, and the number of possible permutations of the events may be huge. These event sequences lead to a large number of GUI states. An event can be tested in many GUI states. The code cannot be used to determine whether these sequences have been tested adequately for a GUI. From the view of the abstract level, the event level has higher position than the code level.

Furthermore, today's GUIs are typically developed using widgets in Toolkits. These widgets have been implemented with a variety of methods and attributes for GUI objects. Many dependencies and implementation details are hidden from developers and testers. Moreover, the source code for these widgets may not always be available for coverage evaluation. Hence, many traditional code-based coverage criteria, such as statement coverage, decision coverage, and path coverage, cannot be directly applied to evaluate the adequacy of GUI tests.

To date, there has been little research on developing coverage criteria for GUI testing. The only reported systematic work on GUI testing is the event sequences based coverage criteria defined by Memon [2001]. They identify events within each component and represent them as an event-flow graph. Three coverage criteria based on events within a component are defined:

Event Coverage: requires the each event in the component be performed at least once.

Event-Interaction Coverage: requires that after an event E has been performed, all events that can interact with E should be executed at least once.

Length- n sequence Coverage: a set P of event-sequences satisfies the length- n event sequence coverage criterion if and only if P contains all event-sequences if length equal to n .

Then, an integration tree is constructed by using the components, and three inter-component coverage criteria were defined:

Invocation Coverage: requires that each restricted-focus event, which opens a modal window, in the GUI be performed at least once.

Invocation-termination coverage: requires that all length 2 event sequences consisting of a restricted-focus event followed by one of the invoked components termination event, which closes a modal window, be tested.

Length- n event-sequence Coverage: requires testing all length- n event-sequences that start with an event in one component and end with an event in another component.

In our GUI testing strategy, the way to represent a GUI system is different from the method described in Memon [2001]. Hence, different event-based coverage criteria are defined for the testing on both tiers respectively. The coverage requires that, first, a set of distinct event sequences are generated to reveal all aspects of a GUI's behavior, and second, we seek a minimal set of these sequences. The following is the definition of the coverage criteria used in our testing strategy.

On the component tier:

A flow graph is developed for each component of a GUI. The nodes in a flow graph represent GUI objects or the decision-making process in terms of the environmental conditions. The edges on a flow graph represent users' GUI activities on GUI objects that trigger the invocation of one object and/or the closure of another. The coverage criteria on this tier requires that the generated test cases cause the execution of all paths in the flow graph. A path is a unique sequence of branches, from the flow graph's entry node to its exit. Two paths are considered equivalent if they differ only in the number of loop traversals. All edges, including those in the flow graph, in the sub-graphs of nodes, and in all models of GUI objects, should be covered at least once in the test cases.

On the system tier:

A finite state model is created to represent the behavior of a GUI. From the entry node to the exit node, each path represents a certain route by which users go through the entire system. However, the number of different paths of a finite state model can be huge or even infinite. Testers are concerned with how to select optimum paths for testing. Actually, for a finite state model, there is a coverage hierarchy, which includes node coverage, transition coverage, transition pair coverage, and longer sequence coverage ([Beizer 1995]). In general, node coverage tells us almost nothing because if the software already has been tested at a lower level, such as unit testing, it is likely that node coverage has been achieved. The essential issue in a finite state model is the transitions among the states. In this thesis, we use transition coverage for the system tier testing, which requires the generation of test cases to cause the execution of all edges in the finite state model. All edges have to be covered at least once for minimum coverage. In practice, for a specific system, longer sequences coverage can be adopted.

Technically, both of the coverage criteria defined in Memon [2001] and in this thesis are event sequence based and consist of the criteria to measure the component level and system level testing. However, there is difference between them. First, for the criteria of Memon, there are many unfeasible event sequences in the total event sequences. In the case study of Memon [2001], 21,659 event sequences were generated up to the length of 3 with 4,189 (or 19.3%) of them being unfeasible. This percentage is likely to significantly vary between systems. Using the number of total event sequences to evaluate the coverage of a test suite can be rather inaccurate. Hence, it is doubtful whether the coverage criteria in Memon [2001] are a set of effective measurement of test adequacy for GUI software. The criteria proposed in this thesis do not have this problem because the logical relationships between GUI objects are involved in the testing models. Second, for the method in Memon [2001], the invocation relationships between components can be over-simplified. Looking back at the example about Print Dialog of Notepad shown in Figure 3.7, we can see that multiple states of the print page range object can cause the event on "OK" button to invoke a message box. But their coverage can be satisfied as long as this invocation is executed no matter which state causes the execution. Hence, some crucial

information of the object will be lost in testing. However, the criteria in this thesis can solve this problem by including all states of GUI objects in the testing models. In Chapter 6, we will further discuss their difference based our experiment.

6. 6. Empirical Investigations and Results

6.1 Overview

The investigation of the effectiveness of a testing strategy lacks formal analytical methods. In general, experimental evaluation is required. In this thesis, two experiments were designed to test the fault-detection capability of the proposed GUI strategy. The first investigation worked on a simplified Notepad program written in Java, which is the example application through the discussion in this thesis. A large number of errors were systematically seeded into Notepad program. Test cases generated according to our testing strategy were executed on the program to see that how many of the seeded errors can be killed. Section 6.2 describes the details of this investigation. The second experiment applied the proposed testing strategy to GUI software called WinCVS. CVS is the Concurrent Version System, which is a very useful open-source version control system. WinCVS is the graphic user interface of CVS. Its stable version has been used everywhere from individual developers to large, distributed teams. However, our test still found some defects in it. Section 6.3 discusses the investigation and the results.

6.2 Empirical Investigation 1

6.2.1 Overview

The first empirical experiment works on a simplified Notepad program written in Java. It is the example application through this thesis to demonstrate the process to conduct the proposed GUI testing approach. Section 2.3 describes the functions of Notepad. Figure 2.2 shows the main window of Notepad. More snapshots on other parts of it are shown in Chapter 3.

The steps to conduct the experiment are:

- *Identify the components and create the corresponding flow-graphs.* After analyzing all available information of the system, eight components are defined. For each of the components, a corresponding flow-graph is created. Chapter 3 shows the details of this process.

- *Construct the Events Sequence Diagram of Notepad on the system tier.* An Event Sequence Diagram is used to represent the finite state machine model of a GUI system. Chapter 4 describes how to perform this process.
- *Generate the test cases according to the coverage defined in Chapter 5.* The test cases were generated with respect to the models on the component tier and the system tier. On the component tier, test cases were designed to cover all branches on the flow-graph of each component; on the system tier, test cases were designed to cover all edges on the event sequence diagram of the system. Finally, 12 test cases for the component tier and 4 test cases for the system tier were generated. The tool that we used to record and play back test script files was Rational Robot. It should be noted that a test case script is an entire sequence of actions required to create a complete user scenario for the system – from start-up through all of the actions and ending with shutdown. A test script can be decomposed into a series of individual test primitives that accomplish specific actions. The primitives can be combined in a specific sequence to provide a test script that verifies a unique usage scenario for the product. For the two-tiered GUI testing approach in this thesis, the test scripts on the component tier may be decomposed into primitives, which can contribute to constructing the test scripts for the system tier testing. Doing so can reduce the effort to develop the test scripts on the system tier.
- *Create the mutants for the experiments.* The conception of mutant here is the same as in mutant testing, in which each program where a fault is seeded is referred to as a mutant. In general, in order to avoid interaction effects between faults, each time one and only one fault is seeded in the program. A detailed description of mutants generated for this experiment will be presented in the next section. Finally, there were 80 mutants built in our investigation.
- *Execute the test cases on the mutants.* Playback all the 16 test script files on the 80 mutants. After executing each test script on a mutant, there are two possible results: “pass” means the test script runs through all the mutant successfully and the outputs are the same as the test outputs; the second is “Fail”, that means that the test script can not run through the mutant

successfully. For a mutant, if all 16 test scripts run through it and all get a "pass" result, this mutant is defined "alive"; if at least one of the test scripts run and get the results of "Fail", the mutant is defined as "killed". The effectiveness of the set of test cases is measured by the number of mutants killed divided by the total number of mutants that were generated. It is common that some mutants are functionally equivalent to the original program. These mutants are defined as "equivalent" mutants. Hence, for the five mutants, we have subsequently analyzed them to determine whether they are "equivalent" mutants. Equivalent mutants are not counted when evaluating the effectiveness of the test cases.

- *Analysis and evaluate the result of the execution.* After the execution of the test cases on all the mutants, the test results are analyzed to evaluate the fault-defect capability of the proposed GUI testing approach.

In this investigation, the quantity and quality of the mutants significantly affect the effectiveness of the evaluation. In order to draw statistical inferences, a large population of mutants was required to be seeded into the system under test. Furthermore, the seeded faults should be similar to practical faults made by designers or developers during the software engineering process. Hence, in order to be systematic and as complete as possible, faults are usually seeded by using mutation operators. However, currently, there is little work on mutant operators specifically to seed GUI faults. Therefore, our solution to produce systematic mutants involving several different mechanisms. The first group of faults came from the faults found during the design and development process of Notepad and our previous programming experience on GUI software. The second group of faults was designed in light of the mutant operators for the Java programming language. The research in this area is fruitful. There are a large number of mutant operators designed according to the possible faults made in Java programs. Actually, only some of the operators are suitable for our experimental purpose because we ignore the faults that are too easy to find. Hence, all mutants are required at least to pass the compilation and be able to run until the main window shows. The next section will discuss the details about the mutant operator selection for the experiment.

6.2.2 Mutant Operators

Mutation testing is a fault-based testing approach, during which faults are introduced into a program by creating many versions of the original program. Each of the faulty programs contains only one fault. Test data are executed on these faulty programs (mutants) with the goal of causing each faulty program to fail. A mutant is killed when a test case causes it to fail (Offutt [1995]). To date it appears to be quite common to use this method to evaluate the effectiveness of test strategies and the adequacy of coverage criteria (Mathur and Wong [1994], Frankl [1996], and Briand [2002]). As discussed in the previous section, in order to make the investigation systematic and as complete as possible, faults are usually seeded by using mutation operators that are usually designed according to a complete classification of faults for a given programming language. However, not all the operators can be applied to the program. When designing the mutants, there are several issues that need to be considered. The first one is the issue of equivalent mutants, which has been discussed previously. Second, some operators generate mutants that may be killed very easily. Although mutants make simple syntactic changes to the program, their semantic impacts can vary greatly. It is necessary to ensure that the seeded faults still allow the program to compile and run.

The GUI system example for this experiment is developed by using Java, which is an object-oriented programming language, very different from the traditional procedural languages, such as FORTRAN and C, as its inheritance, polymorphism, and dynamic binding. All these features allow new kinds of faults during development, some of which cannot be modeled by traditional mutation operators. Combining the work of Harrold and Rothermel [1994], Ma *et al* summarized a comprehensive set of faults for Java, which gave us insight into designing mutant operators (Ma, Kwon and Offutt [2002]). They define four levels of faults:

- (1) *Intra-method level faults*, which occur when the functionality of a method is implemented incorrectly;
- (2) *Inter-method faults* and (3) *intra-class faults*, which are made at the interaction between pairs of methods of a single class or between pairs of methods that are not part of a class;

- (3) *Inter-class faults* that occur due to the object-oriented specific features such as encapsulation, inheritance, polymorphism, and dynamic binding.

On the intra-method level, traditional mutation operators for procedural program and adaptable for Java can be found in Alexander *et al* [2002]; on inter-method and intra-class levels, the mutant operators proposed in Delamaro *et al* [2001] were used in our experiment; and on inter-class level, Ma, Kwon and Offutt [2002] provided six groups of mutant operators.

It should be noted that the Notepad application, used for our evaluation, is not a very complex system involving many coupling relationships and deep inheritance levels. Hence, not all the mutant operators in the four categories discussed previously can be applied. After analyzing all the mutant operators and the Notepad program, we selected eleven mutation operators, applied them to the program, and finally created 74 mutants for investigation. Adding 6 mutants designed by seeding the naturally occurring faults found during development, the total of 80 mutants were created. The appendix of this thesis gives detailed descriptions of all the mutants. Table 6.1 shows the names and concise descriptions of all the mutant operators. Some of the operators can be used on both *intra-method level* and *inter-method level* because when they are applied to different kinds of variables or constants, faults on different levels can be seeded. In order to avoid interactions between the faults seeded in the program, we have to create a mutant for each fault. Usually, running a test case on a mutant takes only several seconds. Adding the time to set up the test, it took a couple of days to execute all test cases on all the mutants.

Table 6.1 The Mutant Operators Used in the Investigation

No.	Mutant Operator	Description
1	Increment the value by 1 "constant"	For scalar constants, increment the value by 1
2	Decrement the value by 1 "constant"	For scalar constants, decrement the value by 1
3	Mutators for Map	Mutators are implemented for the <i>Map</i>

		object in Java.
4	Change the value the constants used in the function	For un-scalar constants, change the value of them.
5	Variable Increment and Decrement Operators – “Variable”	For scalar variables, apply increment or decrement to them.
6	Unary Operator Insertion	Insert an arithmetic, logical, or bit negation operator before the use of each variable and constant.
7	Replace Return Statement Operator	Perturb the value returned by <i>return</i> statements
8	Replace the Operators	Replace the operators that influence the value of non-interface variable or interface variable.
9	Inheritance- IPC	Explicit call of a parent’s constructor deletion
10	Java Features – JTD	<i>This</i> keyword deletion
11	Common Programming Mistakes - EOC	Reference comparison and content comparison replacement

6.2.3 Test Execution Results and Analysis

Once the mutants were created, the test cases generated on the component and the system tiers can be executed. In order to save time and energy, the test cases were recorded as testing scripts and then played back on the 80 mutants using Rational Robot. It is a complete set of components for automating the testing of Microsoft Windows client/server and Internet applications running under Windows NT 4.0,

Windows XP, Windows 2000, and Windows 98. With this tool, the time we spent on recording the scripts and playing them back on all mutants was greatly reduced. Table 6.2 shows the results of executing all the test cases on each of the mutants. In the table, the columns TC1 to TC12 represent the test cases on the component tier; the columns TS1 to TS4 represent the test cases on the system tier; and each row represents one mutant. The * means that the test case evaluated to "fail". Table 6.3 summarizes the data shown in Table 6.2.

Table 6.2 The Result of The Experiment

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	TC12	TS1	TS2	TS3	TS4
Mutant1																
Mutant2						*										
Mutant3					*											
Mutant4																
Mutant5															*	*
Mutant6													*			
Mutant7		*		*					*	*			*	*	*	*
Mutant8		*		*					*	*		*	*	*	*	*
Mutant9	*		*													
Mutant10	*		*													
Mutant11		*		*					*	*			*	*	*	*
Mutant12		*		*		*			*	*			*	*	*	*
Mutant13		*		*					*	*			*	*	*	*
Mutant14													*			
Mutant15		*							*	*			*	*	*	*
Mutant16										*						
Mutant17	*		*													
Mutant18				*												
Mutant19					*									*		
Mutant20								*								
Mutant21																
Mutant22																
Mutant23						*										
Mutant24						*										
Mutant25						*										
Mutant26																*
Mutant27																
Mutant28																
Mutant29																*
Mutant30																*

Mutant31															*
Mutant32															*
Mutant33															
Mutant34					*										
Mutant35												*			
Mutant36															
Mutant37					*										
Mutant38												*			
Mutant39												*			
Mutant40	*		*	*	*			*	*			*	*	*	*
Mutant41	*	*	*	*	*			*	*	*		*	*	*	*
Mutant42							*					*		*	
Mutant43							*					*		*	
Mutant44												*			
Mutant45		*						*	*			*	*	*	*
Mutant46	*		*	*	*			*				*	*	*	*
Mutant47			*	*				*	*			*	*	*	*
Mutant48															
Mutant49	*	*	*	*											
Mutant50	*		*		*										
Mutant51	*		*		*										
Mutant52		*					*								
Mutant53		*													
Mutant54		*													
Mutant55		*						*				*	*	*	
Mutant56					*							*			
Mutant57	*		*	*	*		*								
Mutant58	*		*	*	*			*				*		*	*
Mutant59												*			
Mutant60												*			
Mutant61					*							*			
Mutant62												*			
Mutant63															
Mutant64	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Mutant65	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Mutant66	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Mutant67															
Mutant68															
Mutant69															
Mutant70															
Mutant71															
Mutant72															
Mutant73															
Mutant74	*		*												

Mutant75				*													
Mutant76					*												
Mutant77							*	*			*	*					
Mutant78	*	*	*			*											
Mutant79		*							*	*	*		*	*	*	*	*
Mutant80	*												*				
Killed /test case	17	18	15	17	10	16	6	9	14	17	5	5	31	17	21	22	

Table 6.3 The Results of the Experiment

Fault	Total Mutants	Passed Mutants	Killed Mutants	Equal Mutants	Killed/(Total -Equal)
Faults found in development	6	2	4	0	66.7%
Faults on intra-method, inter-method and intra- class levels	54	7	47	3	92.2%
Faults on inter-class level	20	8	12	7	92%
Sum	80	17	63	10	90%

Upon analyzing the data from the table, the following behavior was discovered:

First, we can see that 90% of the mutants were killed. We analyzed the faults seeded in the remaining "alive" mutants and found that among the seventeen "alive" mutants, ten of them were "equivalent" mutants that operated as the same as the original program and the other seven ones only changed the appearances of certain GUI widgets instead of the functional behavior of the application. For example, one bug with the Print Dialog is when users relocate the dialog, its background area does not repaint properly. Other instances include changing a modal dialog to a modeless dialog or the incorrect layout of GUI objects, etc. No test cases were designed to verify these non-functional features of GUI objects, thus such bugs were missed by the test. Another kind of "alive" mutants are those for which the effects caused by the seeded faults were transient; as long as the user invokes another operation, the faults did not exist anymore. Hence, it was difficult to catch them during the testing.

According to our knowledge, these problems are difficult to be found even by other current approaches to GUI testing. Therefore, our experiment results have led us considering that the approach to solve the difficulties of GUI testing is not a single method. We believe that the integration of some specific GUI testing heuristics may be the most viable route forward. For example, <http://members.tripod.com/~bazman/checklist> provides a long checklist for GUI testing, which covers many of the deficiencies.

Second, the last row of Table 6.2 shows the number of the mutants killed by each of the test cases, which fluctuates greatly from 5 to 31. Comparing the execution results on the two tiers, we found that the test cases on the system tier killed more mutants than the test cases on the component tier. This is in compliance with our expectations, because the test cases on the system tier involve more features of the application than the component one. For the test cases on the component tier, the number of the killed mutants has a close relationship with the number of the lines of code for the associated components, since the seeded faults were distributed relatively evenly over the program, if a component involves more lines of code, it has the potential to have more bugs and its test case is likely to find more faults. Hence, the numbers of killed mutants for the test cases on the component tier do not reveal their exact fault-detection capabilities. Moreover, the test cases on the two tiers cannot be replaced by each other. Although, most mutants can be found by the test cases on both tiers, there are a number of mutants that can only be killed by the component tier test cases or the system tier test cases. Especially for some inter-method or intra-class faults, it is impossible to find out such faults on the component tier. Only the test cases on the system tier can kill them.

In Chapter 5, we have introduced the event sequences based coverage criteria for GUI testing defined by Memon [2001] and technically compared their criteria with the coverage criteria defined in this thesis. Measuring the test cases generated according to the current approach against the coverage defined in Memon [2001] can give us more insight into our testing strategy. This process included identifying the components and events of the Notepad application by the method of Memon [2001], creating event flow-graphs, and computing the coverage metrics against the total number of event-sequences for each component. The maximum length of event

sequences used is up to 3. Table 6.4. shows the results in detail. The columns 1' and 2' represent the invocation and invocation-termination coverage respectively. The definitions of invocation and invocation-termination coverage are described in Section 5.2.

After getting the total number of event-sequences for each component, we can use the algorithms in Memon [2001] to evaluate the coverage of our test suite. Table 6.4 shows the total number of event sequences of Notepad's components. Table 6.5 shows the coverage metrics of our test cases, which are represented by the percentages of total event-sequences covered by the test cases. The column 1' and 2' in Table 6.4 are the invocation and the invocation-termination events respectively. And the column 1' and 2' in Table 6.5 are the corresponding coverage. The results of the comparison are not surprising. The columns 1' and 1 of Table 6.5 have all 100% coverage because all components and their GUI objects are triggered at least once. Considering the case study shown in Memon [2001], we can find that for the test cases generated by the Plan system, there is always a small portion of the events never used by the planner since they did not contribute towards a goal. This has been solved by the approach in this thesis.

Table 6.4. The Total Number of the Event-sequences in Notepad

Component name	Event-sequences Length				
	1'	2'	1	2	3
Main	1	1	19	55	330
File open	1	2	9	63	441
File save	1	2	9	63	441
Print	1	2	8	48	288
Font	1	2	5	9	45
Find word	1	2	3	3	-

Table 6.5. The Percentage of the Event-sequences Covered

Component name	Event-sequences Length				
	1'	2'	1	2	3
Main	100	100	100	43.6	7
File open	100	50	100	12.7	2
File save	100	50	100	12.7	1.6
Print	100	50	100	30	3.5
Font	100	100	100	55	8.9
Find word	100	100	100	66.7	-

Looking at the columns for event sequence equivalent to 2 and 3 of Table 6.5, we note that only a small percentage of length 2 and 3 event sequences were covered. Moreover, we can predicate that the percentages for the longer sequences will rapidly decay because the total number of longer event sequences grows dramatically. But our test cases found 90% of the faults seeded. Thus, there comes a question that whether the test cases that fully satisfy Memon's coverage can reveal the remaining 10% faults. The answer is "No.". As discussed previously in this section, the remaining 10% faults were concurring non-function features of GUIs, which are challenges to the methods by Memon [2001] and White [2000]. Generally, under a tight time and cost budget, finding the most effective test cases is required. In some circumstances, testers have to discard many low value cases that imply high cost. From the comparison we can see that our test cases and coverage criterion has higher effectiveness.

6.3 Empirical Investigation 2

6.3.1 Introduction

The second experiment worked on WinCvs, which is the graphical user interface of CVS. CVS stands for Concurrent Versions System. It is a version control system that

has been developed in the public domain since 1986. Using it, developers can record the history of their source files. For example, bugs sometimes creep in when software is modified, and developers might not detect the bug until a long time after they make the modification. With CVS, developers can easily retrieve old versions to see exactly which change caused the bug. This can sometimes be a big help during software development.

However, the biggest problem with CVS is that it uses a command line interface. Since most developers prefer a graphical user interface, several graphic front ends to the CVS core have been developed. The best one available for Windows NT/98 is WinCvs. Figure 6.1 shows the main window of WinCvs Version 1.0. More information is available on the web site at www.wincvs.org. This stable version has been used everywhere from individual developers to large, distributed teams. Appendix B describes the main tasks implemented in WinCvs 1.0. Our approach in this second experiment is to apply the proposed testing strategy to WinCvs to see whether it interacts correctly with both the user and the underlying application. The results show that our test cases still found some faults and defects.

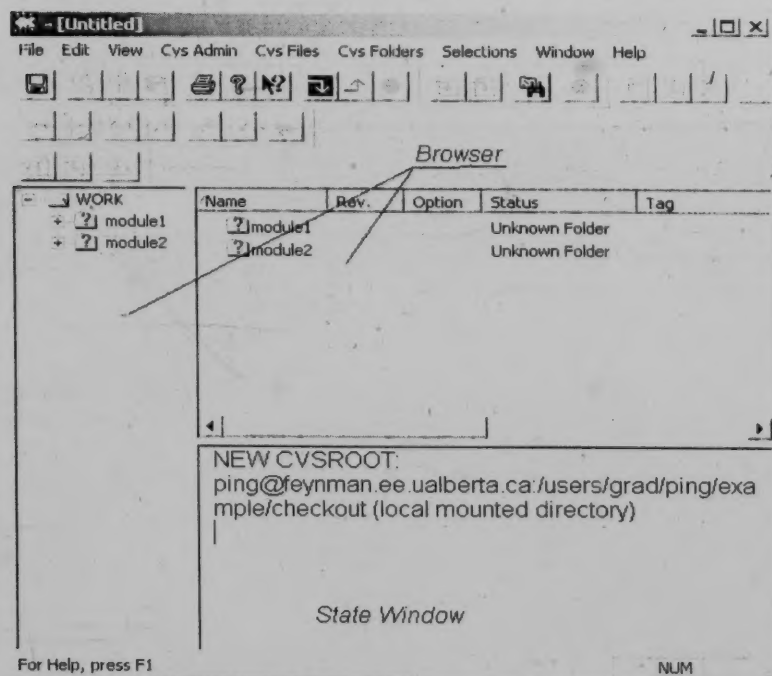


Figure 6.1 The Main Window of WinCvs

6.3.2 Results and Discussion

Different from the commercial software projects, WinCvs is open source project developed by a group of people all around the world. There is no formal requirement specification available for our experiment. Hence, we tried to collect as many resources about WinCvs as possible. The resources include the user manuals of WinCvs, the user manuals of CVS that, to some extent, can be regarded as the requirement specification for WinCvs, discussions between users and the developers published on their website and our direct interaction with WinCvs. The experiment working conditions are:

WinCvs: version 1.0

CVS core version: 1.1

Operation system: Windows 2000

The entire testing took one person four weeks to implement. One week was spent to study CVS system and the operations implemented by WinCvs; two weeks to analyze all GUI objects of WinCvs, identifying the responsibilities and designing the test cases; then the last week to apply the test cases to the WinCvs system and analyze the testing result. Twenty seven responsibilities were identified and the corresponding components were created. Finally, 30 test cases were designed on the component tier and 4 test cases on the system tier. In practice, although we tried to cover all functions implements in WinCvs, for the limited time and technique conditions, this test was not complete. But it still discovered many faults and defects in WinCvs listed in the following table.

Table 6.6 The Faults Found in WinCvs

No.	Related Function	Description
1	Set a work area	When users set a work area on a network place, in the browser for folder, users try to create new folder in a invalid disk, two error message boxes show. The first one is not relative with this situation.

2	Editing a file	After a file in the work area has been modified, users select "unedit selection button". WinCvs prompts a message box to users to select to revert the modification or not. But it doesn't change the property of the file as it should do. Then when users click "edit select button" and click "unedit select button" again, WinCvs does not prompt any message box and it changes the property of the file. WinCvs shows inconsistent behavior in this situation.
3	Check the status of a file	When checking the status of files in a repository, the importing time of the files shown in the status window are different from the actual importing time and the operating system time. And there is nowhere to adjust the time system in WinCvs.
4	Check the difference before committing	When there is no difference between the compared revisions, the exit code is 0; when there is difference between the two revisions, the code is 1. When CVS diff aborts, the exit status code is 1 too. This situation should be assigned a different exit code to avoid confusions.
5	Browser	The browser does not show the directories normally.
6	Print	The print function does not work at all
7	Print preview	Starting the print preview multiple times can cause WinCvs to hang up.

8	Lock files	When a user locks a file, it should be able to be unlocked by the same user after using the file. But in WinCVS, users cannot unlock the files locked by themselves.
9	Create a tag of module	In "create a tag" dialog, if users do not select the check-box "Tag also the attic folder", the attic folder should not be tagged. But WinCvs tag it too.

7. Conclusion and Future Works

This thesis presents a systematic and effective approach to test GUI software, utilizing two different testing strategies: the component tier and the system tier testing strategies. Additionally, corresponding coverage criteria are defined as well. Two empirical investigations were designed to verify the usefulness and effectiveness of this two-tiered testing methodology for GUI testing.

From the empirical investigation we can draw the conclusion that:

1. The proposed two-tiered testing approach covers most of the characteristics of GUIs and the test cases generated are effective at fault detection.
2. GUI testing is a very complex problem and finding a thorough and perfect solution is extremely difficult. An individual method as the "silver bullet" for it seems impossible. Combining the strength of multiple methods may conquer the difficulty of GUI testing.

Finally, our proposed methodology has two additional benefits, specifically:

1. The decomposition of the GUI system into responsibility components could be more flexible under cost and time pressure often found in industrial testing situations. This flexibility allows us to tailor the amount of effort involved in the testing to accommodate these economic requirements. This value-oriented approach to test case management has obvious parallels with the use of operational profiling Musa [1998].
2. Our test strategy also facilitates regression testing. During the life cycle of a software application, modification is inevitable. After fixing a recognized problem or adding new features, all possible involved parts should be retested for inadvertently introduced errors. Our testing method has broken the entire system into responsibility components, and to locate the modified component is now straightforward and convenient. This sub-division again allows for mechanisms to minimize the amount of testing - restricting it only to affected components.

Several areas of possible future research are presented below:

1. As discussed in Chapter 2, the process to represent the testing model and generate test cases is done manually. Our future work includes to make this testing process automatically and the discovery of efficient algorithms to convert the model into an expression, which can be transferred to test scripts that can be used by commercial test tools.
2. Another future research direction is to explore the possibility of using the proposed approach for software other than GUIs, such as web-based software, which has many parallels but is significantly different with GUI software.

In today's fast-paced software development environments, all participants must work together to ensure a product to be successfully accomplished. Designers, developers, testers and managers must cooperate. Planning for testability during the design and coding phases of a software product can provide significant efficiency in testing the product and has a very small cost and schedule impact on the development processes. The following is some recommendations for designers and developers aimed at improving testability of GUI software.

- Designers should make a clean separation of the GUI and the algorithmic aspects. The behavior of GUI objects should be observable and controllable.
- Unless there is a strategic or structural reason, standard Windows controls should be used. Standard methods and properties should be used when the test tool works correctly with them.
- Navigation between windows should be hierarchic to minimize the number of windows that might be open at once and to reduce the number of paths through the system.
- Unless there is an impact on usability, dependencies between objects on windows should be avoided or circumvented by splitting user transactions into multiple modal windows.

Bibliography

- Albin [2003] Stephen T. Albin, The Art of Software Architecture: Design Methods and Techniques. John Wiley & Sons, 2003.
- Apfelbaum[1997] Larry Apfelbaum, John Doyle. Model Based Testing. Teradyne Software & Systems Test 1997
- Alexander [2002] Roger T. Alexander, James M. Bieman etc. Mutation of Java Objects Proc. IEEE Int. Symp. Software Reliability Engineering (ISSRE) 2002.
- Apfelbaum[1998] Larry Apfelbaum, John Schroeder. Reducing the Time to Thoroughly Test a GUI. Teradyne Software & Systems Test, 1998
- Beizer [1995] Boris Beizer, Black-Box Testing: Techniques for Functional Testing of Software and Systems, John Wiley & Sons, 1995
- Belli [2001] Fevzi Belli: Finite State Testing and Analysis of Graphical User Interfaces, Proc. ISSRE 2001, IEEE Comp. Press, pp 34-43
- Binder [2000] Robert V. Binder. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison Wesley. 2000
- Bochmann [1994] G.V.Bochmann, A.Petrenko, Protocol Testing: Review of Methods and Relevance for Software Testing, In Proceedings of 1994 International Symposium on Software Testing and Analysis pages 109-124 1994
- Briand [2002] L.C.Briand, Y. Labiche, Y. Wang Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts Software Quality Engineering Laboratory, Carleton University 2002.
- Chow [1978] Tsun Chow. Testing Software Design Modeled by Finite-State Machines. IEEE Trans. On Software Engineering, Vol SE-4, #3, pp 178-187, May 1978.

- Delamaro[2001] Marcio E. Delamaro, Jose C. Maldonado, and Aditya P. Mathur, Interface Mutation: An Approach for Integration Testing. IEEE Transactions on Software Engineering, 27(3):228-247, March 2001
- Dwyer [1997] Matthew B. Dwyer, Vicki Carr, Laura Hines. Model Checking Graphical User Interfaces Using Abstractions, Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), September 1997
- El-Farand [2001] Ibrahim K. El-Farand and James A. Whittaker. Model-based software testing. Encyclopedia of Software Engineering, 2nd edition, John Wiley & Sons.2002.
- Fewster&Graham[1999] Mark Fewster and Dorothy Graham: Software Test Automation. Addison-Wesley,1999
- Frankl [1996] P. G. Frankl, S. N. Weiss and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness", Journal of Systems and Software, vol. 38 (3), pp. 235-253
- Galitz [2002] Wilberta O. Galitz. The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques. John Wiley & Sons, 2002.
- Gerrard [1997] Paul Gerrard, Testing GUI Applications, EutoSTAR'97 24-28 November 1997 Edinburgh UK.
- Gill [1962] A. Gill Introduction to the Theory of Finite-State Machines, McGraw-Hill, 1962
- Harrold [1994] Mary Jean Harrold, Gregg Rothermel, Performing data flow testing on classes, Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'94) pp. 494-505, March 1994
- Lee [1996] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceeding IEEE*, 84(8):1090–1123, Aug. 1996.

- Hetzel [1998] Bill Hetzel, The Complete Guide to Software Testing. Second Edition. John Wiley & Sons, 1998
- Hopcroft&Ullman[1979] John E. Hopcroft and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company. 1979.
- IPL [1997] IPL Information Processing Ltd. Structural Coverage Metrics: Their Strength and Weakness. 1997
- Kaner ,etc [1999] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. Testing Computer Software. John Wiley & Sons, 1999.
- Kepple[1994] Laurence R. Kepple The Black Art of GUI Testing, 40,42,46, Dr. Dobb's Journal, Feb. 1994
- Leverson&Turner[1993] Nancy Leverson and Clark S. Turner. An Investigation of the Therac-25 Accidents. IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41
- Ma [2002] Yu-seung Ma, Yong-Rae Kwon, Jeff Offutt, Inter-Class Mutation Operators for Java Thirteenth International Symposium on Software Reliability Engineering, Annapolis, MD, November 2002.
- Mathur[1994] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria", Journal of Software Testing, Verification, and Reliability, vol. 4(1), pp. 9-31, 1994
- Memon [2001] Atif Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage Criteria for GUI Testing, 8th Europe Conference and 9th ACM SIGSOFT Foudation of Software Engineering (FSE-9), Austria, September 10-14, 2001
- Memon [2000a] Atif Memon, Martha E. Pollack, Mary Lou Soffa, Automated Test Oracles for GUIs. The Proceedings of the Eighth International Symposium on the Foundations of Software Engineering, in San Diego, CA. Nov. 2000

- Memon [2000b] Atif Memon, Martha Pollack and Mary Lou Soffa, "Plan Generation for GUI Testing", the 5th International Conference on Artificial Intelligence Planning and Scheduling, Breckenridge, CO, pp. 226-235, April 2000
- Musa [1998] John Musa, Software Reliability Engineering. McGraw-Hill, 1998
- Myers [1979] Glenford J. Myers, The Art of Software Testing, Wiley 1979
- Myers [1993] Brad A. Myers, Dan R. Olsen. User Interface Tools. In Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems - Adjunct Proceedings, Tutorials, page 239 1993
- Myhill [1957] Myhill, J. Finite Automata and the Representation of Events, Wright Air Development Command, TR 57-624, pp.112-137 1957.
- Offutt [1995] A. Jefferson Offutt, "A Practical System for Mutation Testing: Help for the Common Programmer" Twelfth International Conference on Testing Computer Software, pages 99--109, Washington, DC, June 1995
- Parnas [1969] David L. Parnas On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System, Proc. 24th National ACM Conf. 379-385, 1969
- Perry [2000] William E. Perry, Effective Methods for Software Testing, Second Edition. John Wiley & Sons, 2000
- Roper [1994] Marc Roper. Software Testing. McGraw-Hill, 1994
- Shehady [1997] Richard Shehady and Daniel Siewiorek. A Method to Automate User Interface - Testing Using Variable Finite State Machines. Proc. Of the 27th Int. Symposium on Fault Tolerant Computing, Seattle, WA, pp 80-88, June 1997

- The. [1992] L. The. Stress Tests for GUI Programs. Datamation, 37-38, Sept. 1992
- Watkins [2001] John Watkins: Testing IT: An Off-the-Shelf Software Testing Process. Cambridge University Press. 2001
- White [1996] Lee J. White, Regression Testing of GUI Event Interactions. Proc. Of the International Conference on Software Maintenance, Washington, DC, pp 350-358, Nov. 1996
- White [2000] Lee White, Husian Almezen Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences, International Symposium on Software Reliability Engineering, San Jose, CA, October 2000, pages. 110-121.
- White [2001] Lee J. White, Husain Almezen and Nasser Alzeidi. User-Based Testing of GUI Sequences and Their Interactions. 12th International Symposium on Software Reliability Engineering (ISSRE'01) November 27 - 30, 2001 Hong Kong, China
- Whittaker [2002] James A. Whittaker, How to Break Software: A Practical Guide to Testing. Addison Wesley, May 2002

Appendix A

This Appendix gives detailed descriptions of the mutants used in our first experiment.

Mutant description:

No.	Mutants Description	Seeding Location	Operator Type
1	When the Print Dialog changes its position, the area covered by it doesn't repaint	Print Dialog	Found during development
2	After using the Find function to find a text in text area, then using the Find Next, it doesn't highlight the found word.	Find and Find Next	Found during development
3	When opening a file, if the file name is invalidate, there is no message for it	Open Dialog	Found during development
4	When opening a new file, the undo function is not undated.	undo and redo	Found during development
5	The first time open notepad, the wrap-line is not set but the text area's wrap-line property is set.	wrapline	Found during development
6	fileName=null; missing	new and save dialog	Found during development
7	//while((nch = in.read(buff, 0, buff.length)) != -1) while((nch = in.read(buff, 1, buff.length)) != -1)	Action.java -- open() Line439	Increment the value by 1
8	//n.getTextArea().append(new String(buff, 0, nch)); n.getTextArea().append(new String(buff, 1, nch));	Action.java -- open() Line440	Increment the value by 1
9	//if(option == 0){ if(option == 1){	Action.java -- new() Line54	Increment the value by 1
10	//if(option == 1){ if(option == 2){	Action.java -- new() Line64	Increment the value by 1
11	//while((nch = in.read(buff, 0, buff.length)) != -1) while((nch = in.read(buff, -1, buff.length)) != -1)	Action.java -- open() Line437	Decrement the value by 1
12	//n.getTextArea().append(new String(buff, 0, nch)); n.getTextArea().append(new String(buff, -1, nch));	Action.java -- open() Line415	Decrement the value by 1
13	after fullDescription = null; add filters.remove(extension.toLowerCase());	ExampleFilefi lter.java Line197	Mutators for map
14	if(!n.getTextArea().getText().equals("")) && change "" to null	Action.java -- new() Line40	change the value the constants used in the function
15	If(!n.getTextArea().getText().equals("")) && change "" to null	Action.java -- open() Line122	change the value the constants used in the function
16	If(!n.getTextArea().getText().equals("")) && change "" to null	Action.java -- exit() Line237	change the value the constants used in the function
17	//if(fileName == null){ if(fileName == ""){	Action.java -- new() Line46	change the value the constants used in the function

18	//if(fileName == null){ if(fileName == ""){	Action.java -- open() Line128	change the value the constants used in the function
19	//if(fileName == null){ if(fileName == ""){	Action.java -- save() Line202	change the value the constants used in the function
20	//if(fileName == null){ if(fileName == ""){	Action.java -- exit() Line242	change the value the constants used in the function
21	//option = JOptionPane.showConfirmDialog(null,"Do you want to save the changes ??"); option = JOptionPane.showConfirmDialog(n,"Do you want to save the changes ??");	Action.java -- new() Line51	change the value the constants used in the function
22	//option = JOptionPane.showConfirmDialog(null,"Do you want to save the changes ??"); option = JOptionPane.showConfirmDialog(n,"Do you want to save the changes ??");	Action.java -- new() Line77	change the value the constants used in the function
23	//n.getTextArea().setText(""); n.getTextArea().setText(null);	Action.java -- new() Line58	change the value the constants used in the function
24	//if(pos==1) if(pos==1)	Action.java -- findNext() Line352	change the value the constants used in the function
25	//pos=n.getTextArea().getText().indexOf(findword,po s+1); pos=n.getTextArea().getText().indexOf(findword,pos);	Action.java -- findNext() Line351	Variable Increment and Decrement Operators
26	//textArea.setLineWrap(false); textArea.setLineWrap(true);	Notepad.Note pad() line363,364	change the value the constants used in the function
27	//n.undoAction.setEnabled(false); n.undoAction.setEnabled(true);	Action.java -- new() Line110	change the value the constants used in the function
28	//n.undoAction.setEnabled(false); n.undoAction.setEnabled(true);	Action.java -- open() Line445	change the value the constants used in the function
29	//n.getTextArea().setLineWrap(true); n.getTextArea().setLineWrap(false);	Action.java -- lineWrap() Line362	change the value the constants used in the function
30	//n.getTextArea().setWrapStyleWord(true); n.getTextArea().setWrapStyleWord(false);	Action.java -- lineWrap() Line363	change the value the constants used in the function
31	//n.getTextArea().setLineWrap(false); n.getTextArea().setLineWrap(true);	Action.java -- lineWrap() Line370	change the value the constants used in the function
32	//n.getTextArea().setWrapStyleWord(false); n.getTextArea().setWrapStyleWord(true);	Action.java -- lineWrap() Line371	change the value the constants used in the function
33	//font.setVisible(false); font.setVisible(true);	Action.java -- font() Line384	change the value the constants used in the function

34	//setEnabled(false); setEnabled(true);	Notepad.UndoAction() line382	change the value the constants used in the function
35	//setEnabled(true); setEnabled(false);	Notepad.UndoAction() line397	change the value the constants used in the function
36	//setEnabled(false); setEnabled(true);	Notepad.UndoAction() line401	change the value the constants used in the function
37	//setEnabled(false); setEnabled(true);	Notepad.RedoAction() line409	change the value the constants used in the function
38	//setEnabled(true); setEnabled(false);	Notepad.RedoAction() line424	change the value the constants used in the function
39	//setEnabled(false); setEnabled(true);	Notepad.RedoAction() line428	change the value the constants used in the function
40	//return true; return false;	ExampleFilefilter.access() Line152	change the value the constants used in the function
41	//If(f != null){ if(f == null){	ExampleFilefilter.access() Line151	unary operator insertion
42	//return cajib; return okjb;	Fonts.getCajib() () Line40	Replace Return Statement operator
43	//return okjb; return cajib;	Fonts.getOkjb() () Line37	Replace Return Statement operator
44	//if(!n.getTextArea().getText().equals("") && !n.getTextArea().getText().equals(fileContent)){ if(!n.getTextArea().getText().equals("") !n.getTextArea().getText().equals(fileContent)){	Action.java -- newW() Line40	Replace the operators
45	//if(!n.getTextArea().getText().equals("") && !n.getTextArea().getText().equals(fileContent)){ if(!n.getTextArea().getText().equals("") !n.getTextArea().getText().equals(fileContent)){	Action.java -- openN() Line119	Replace the operators
46	//If(!n.getTextArea().getText().equals("") && !n.getTextArea().getText().equals(fileContent)){ if(!n.getTextArea().getText().equals("") !n.getTextArea().getText().equals(fileContent)){	Action.java -- exit() Line240	Replace the operators
47	//if(extension != null && filters.get(getExtension(f)) != null) if(extension != null filters.get(getExtension(f)) != null)	ExampleFilefilter.access() Line155	Replace the operators
48	//if(i>0 && i<filename.length()-1) if(i>0 i<filename.length()-1)	ExampleFilefilter.getDescription() Line172	Replace the operators
49	//if(fileName == null) if(fileName != null)	Action.java -- newW() Line46	Replace the operators
50	//if(option == 0){ if(option != 0){	Action.java -- newW() Line54	Replace the operators

51	//if(option == 1){ if(option != 1){	Action.java -- new() Line63	Replace the operators
52	//if(option == 0){ if(option != 0){	Action.java -- new() Line82	Replace the operators
53	//if(option == 1){ if(option != 1){	Action.java -- new() Line89	Replace the operators
54	//while(n.getTextArea().getText().indexOf(findword) == -1) while(n.getTextArea().getText().indexOf(findword) != -1)	Action.java -- finD() Line330	Replace the operators
55	//if(pos== -1) if(pos!= -1)	Action.java -- finDNext() Line342	Replace the operators
56	//if(returnVal == JFileChooser.APPROVE_OPTION) if(returnVal != JFileChooser.APPROVE_OPTION)	Action.java -- open() Line426	Replace the operators
57	//if(returnVal == JFileChooser.APPROVE_OPTION) if(returnVal != JFileChooser.APPROVE_OPTION)	Action.java -- saveAs() Line486	Replace the operators
58	//while((nch = in.read(buff, 0, buff.length)) != -1) while((nch = in.read(buff, 0, buff.length)) == -1)	Action.java -- open() Line437	Replace the operators
59	//if(undo.canRedo()){ if(!undo.canRedo()){	Notepad.Note pad() line423	unary operator insertion
60	//if(undo.canUndo()){ if(!undo.canUndo()){	Notepad.Note pad() line396	unary operator insertion
61	//super("Undo", new ImageIcon("images/undo.gif")); super("Undo", new ImageIcon("images/undo.gif"));	Undoaction() line381	Inheritance-IPC
62	//super("Redo", new ImageIcon("images/redo.gif")); super("Redo", new ImageIcon("images/redo.gif"));	Redoaction() line408	Inheritance-IPC
63	//super(parent,modal); super(parent,modal);	Fonts.java -- Fonts() Line44	Inheritance-IPC
64	//this.n=n; n=n;	Action.java -- action() Line26	Java features- JTD
65	//this.n=n; n=n;	Center.java- Center() Line10	Java features- JTD
66	//this.f=f; f=f;	Center.java- Center() Line13	Java features- JTD
67	//this.filters = new Hashtable(); filters = new Hashtable();	ExampleFileF ilter.java -- ExampleFileF ilter() Line83	Java features- JTD
68	//this.componentToBePrinted = componentToBePrinted; this.componentToBePrinted = componentToBePrinted;	print.java- open() Line15	Java features- JTD
69	//private Hashtable filters = null; private Hashtable filters;	ExampleFileF ilter.java -- Line71	Java features- JTD
70	//private String description = null; private String description;	ExampleFileF ilter.java -- Line72	Java features- JTD
71	//private String fullDescription = null; private String fullDescription;	ExampleFileF ilter.java -- Line73	Java features- JTD

72	//private String fileContent = null; private String fileContent;	Action.java -- Line17	Java features- JID
73	//private String fileName = null; private String fileName;	Action.java -- Line18	Java features- JID
74	//if(fileName == null) if(fileName.equals(null))	Action.java -- new() Line46	Common Programming Mistakes - EOC
75	//if(fileName == null) if(fileName.equals(null))	Action.java -- open() Line128	Common Programming Mistakes - EOC
76	//if(fileName == null) if(fileName.equals(null))	Action.java -- save() Line205	Common Programming Mistakes - EOC
77	//if(fileName == null) if(fileName.equals(null))	Action.java -- exit() Line241	Common Programming Mistakes - EOC
78	//if(!n.getTextArea().getText().equals("")) && !n.getTextArea().getText().equals(fileContent)){ if(!n.getTextArea().getText().equals("")) && !n.getTextArea().getText().equals(fileContent)){	Action.java -- exit() Line238	Common Programming Mistakes - EOC
79	//if(!n.getTextArea().getText().equals("")) && !n.getTextArea().getText().equals(fileContent)){ if(!n.getTextArea().getText().equals("")) && !n.getTextArea().getText().equals(fileContent)){	Action.java -- open() Line122	Common Programming Mistakes - EOC
80	//if(!n.getTextArea().getText().equals("")) && !n.getTextArea().getText().equals(fileContent)){ if(!n.getTextArea().getText().equals("")) && !n.getTextArea().getText().equals(fileContent)){	Action.java -- new() Line40	Common Programming Mistakes - EOC

Appendix B

This appendix gives some important terminologies in WinCVS and describes the main functions implemented in it.

Terminology

In an attempt to avoid confusion, we provide a short list of the most essential terms used in the CVS, and also in WinCvs. Making yourself familiar with these words will help you understand the following function descriptions.

Repository

The CVS repository stores a complete copy of all the files and directories, which are under version control.

Work area

Work area refers to a folder that you will use as the root of your local work area. You can manage multiple work areas with WinCvs.

Checkout

Describe the first retrieval of an entire module from the repository.

Commit

Send your modifications to the repository.

Export

Extract an entire module from a repository without CVS administrative files: the exported modules will not be under CVS control.

Import

Create a new module in the repository by sending an entire directory structure.

Module

A directory hierarchy. A software project normally exists as a single module in the repository.

Release

The version of an entire product.

Revision

The version of a single file.

Tag

A symbolic name given to a set of files at a specific time of development.

Update

Get other users' modifications from the repository. Update the local copy only.

The Functions of WinCVS

- Starting a New Module

Getting a project under version control is called *importing* in CVS terminology. As the name implies, you should have something for the CVS server to import before thinking version control.

CVS lacks good support for restructuring a project file hierarchy, so you save yourself a lot of trouble if you spend some time planning before importing a new module.

The basis for the import operation is a "clean" directory structure. By clean we mean that files which need no versioning (compiler generated files, backups etc.) are removed. This is particularly important if your project has been going on for a while. You may have several files in your hierarchy that you do not want to put under version control, but that you want to keep where they are anyway. In that case you have to move them out before importing, and move them back afterwards.

Be aware that CVS treats empty directories as non-existent. If you want to add a directory in which you have neither files nor subdirectories, you will have to create a dummy file in it. We suggest that you create a file named README.txt, which contains a short description of what this directory is for.

Once you have a directory structure that contains only files you want to have in the repository, use the *CVS Admin -> Import module* menu and the *Import settings* dialog to import your entire directory hierarchy to the CVS server. Note that nothing will be changed in your local files during the import operation. This means that your local copy will not be under version control after the import.

- Getting a New Module from CVS

To obtain a module from the CVS server for the first time is known as a *checkout*. Checking a module out from the repository gives you a local copy of the directory hierarchy that makes up the module. To perform a checkout using the CVS Admin -> Checkout module menu and Checkout setting dialog.

If you state that Checkout read-only should be in use, you will have to tell WinCVS which files you intend to edit before editing them. This may be cumbersome, but it enables other developers to track who is currently editing given files.

- Getting Other People's Changes from CVS

Occasionally you may want changes done by others to get incorporated in your local working copy. The process of getting changes from the server to your local copy is known as *updating*. Updating may be done on single files, a set of selected files, or recursively on entire directory hierarchies. To update the Update selection menu item.

Changes done by others will be merged into your files, keeping any changes you may have done to the same files. The repository is *not* affected by an updating. If you receive reports of conflicts during the update, please read the next section.

- Resolving Conflicts

Once in a while, the CVS server will report a *conflict* when you update your files from the repository. A conflict occurs when two or more developers have changed the same few lines of a file. As CVS knows nothing of your project, it leaves resolving the conflicts to the developers. Whenever a conflict is reported, you should open the file in question, and search for lines starting with the string <<<<<<<. The conflicting area is marked like this:

```
<<<<<<<filename
your changes
=====
code merged from repository
>>>>>>> revision
```

You should decide what the code should look like, do the necessary changes, remove the CVS markup, and commit your modifications to the repository.

- Sending Your Changes to CVS

Making local modifications available in the repository is known as *committing* the changes.

Before committing, you should do an update to make sure there are no conflicts.

Select the directory, file or files that you want to commit. Choose the Commit selection menu item and the Commit settings dialog to commit your changes.

Please note that committing changes will not automatically add new files that you have created to the repository.

Viewing Changes

WinCvs may be used for viewing status, logs, diffs etc, of files and directories.

Select the directory, file or files that you want to know more about.

Click the right mouse button on the selection, and choose one of Diff selection, Log selection, Status selection or Graph selection.

Here is a short explanation on what output you might expect from the various status commands.

Diff

Let you compare your local copy of a file with any revision of the same file in the repository. It also lets you compare different revisions within the repository. The output consists of lines starting with < or >, symbolizing lines that should be removed or added respectively, to go from your revision to the one you compare against.

Log

Show log messages, dates, tags, authors, etc. for all revisions of the given files (unless you limit what revisions to display).

Status

Display the modification status of the given files, i.e. whether the files are modified locally or in the repository. It also shows both the local and the repository revision numbers, and tags, if any.

Graph

A rather cool feature that shows the revision graph for a single file. Particularly useful when you have one or more branches in the file's revision history. The local revision is marked by a document icon. The graph lets you select two revisions (using the shift button for some reason) and do a diff between them.

- Tagging Files with a Label

At a given stage of development, giving one or more files a common label to refer to their revisions is known as *tagging* those files. Tagging is typically used on entire modules, so that the current state of the module can be reconstructed in the future. This kind of tagging should always be done on project deliverables, and before starting major changes.

To tag one or more files or directories with a label using the Tag selection -> Create a tag menu item and Create tag setting dialog.

Note that CVS is quite restrictive when it comes to what characters a tag may contain. A tag must start with a letter, and may contain letters, digits, "-" (dash) and "_" (underscore) only. In particular, this means no dots, and no spaces. If you want to include version numbers in a tag, replace the dots with dashes. Two tag names are reserved, as they have special meaning in CVS: "HEAD" refers to the most recent version available in the repository, while "BASE" is the revision you last checked out into the local directory.

- Adding Files and Directories

When you create new files that you want to include in the repository, you must tell CVS to handle the files. If the directory containing the files is not under CVS control, you will have to add it before adding the files. To add files or directories using the Add selection or Add selection binary menu item and Select files dialog or Select directory dialog.

If you want to add entire directory hierarchies instead of a few files, the above technique becomes cumbersome as the *add* operation is not recursive. In this case you should rather use *import*.

WinCvs will import your entire directory hierarchy to the CVS server, under the given module. Note that nothing will be changed in your local files during the import

operation. This means that your local copy will not be under version control after the import.

- Removing Files and Directories

To remove files, you first schedule the files for removal, and then commit the change using the Remove selection menu item.

As the files are only *marked* for removal, you have to commit them to remove them from the repository. The files will now be removed from the repository. Note that files are not physically removed, but rather marked as "dead". By this way it will still be possible to retrieve the files if you choose to check out an old version of the module.

Removing directories is another story. CVS will optionally remove empty directories when you *update* one of its parent directories. If you want to get rid of an empty directory, do the following:

Select the parent directory of the empty directory you want to remove and then use the Update selection menu item to remove the directory.

- Moving or Renaming Files and Directories

Operation of moving or renaming files or directories is not available in CVS. This is one of CVS's shortcomings. To simulate moving or renaming, you have to combine remove and add operations.

- Branching

One of the features of version control systems is the ability to isolate changes onto a separate line of development. This line is known as a *branch*. To create a branch using the Tag selection -> Create a branch menu item and Create a branch setting dialog. **Note:** The branch is created *in the repository only*. To start working on the newly created branch, you have to do what is described in *Selecting a Branch to Work on*.

- Selecting a Branch to Work on

To start working on a branch instead of the default development line, you have to bind your local copy to the branch. This is needed to make sure that actions such as updates, commits etc. work on the branch rather than on the main line of development.

To move your local copy to another branch, do the following:

Select the top-level directory of the project. (If you know exactly what directories and files are part of the branch, you may select these instead.)

Click the right mouse button on the selection, and choose the Update selection menu item.

Make sure Create missing directories that exist in the repository is checked.

Select the Sticky options tab.

Click the Retrieve rev./tag/branch checkbox.

In the Retrieve rev./tag/branch input field, enter the tag name of the branch you want to switch to.

Press the OK button.

WinCVS will now do the necessary updates to your working copy and move it to the desired branch. The updating may include adding or removing files.

CVS puts what is known as *sticky tags* on the files that are affected by the branch. You may view these sticky tags by issuing a *status* command on the files, as described in Viewing Changes. To remove the sticky tags and thus go back to the main development line, follow the description in Going Back to the Main Line of Development.

- Merging from a Branch

When you are satisfied with the changes you have done on a branch, you may want those changes to be available on the main line of development. Incorporating changes from one branch to another is known as *merging*. To merge from a branch, do the following:

Move your local copy to the branch you want to merge the changes into. See Selecting a Branch to Work on or Going Back to the Main Line of Development.

Select the top level directory of the project. (If you know exactly what directories and files are part of the branch, you may select these instead.)

Click the right mouse button on the selection, and choose the Update selection menu item.

Make sure Create missing directories that exist in the repository is checked.

Select the Merge options tab.

Click the Only this rev./tag radio button.

In the Only this rev./tag input field, enter the tag name of the branch you want to merge from.

Press the OK button.

Any changes on the branch will now be merged into your local copy. You will probably also want to commit the merged files back to the repository, as described in *Sending Your Changes to CVS*.

The merge given above will try to merge changes from the start of the branch. If you do the operation a second time (to merge changes done to the branch after the last merge), merging from the start of the branch is not what you want, and it will most likely get you into trouble. To get around this problem, you should give the branch a new tag after every merge, and use the new tag when naming the branch for subsequent merges.